Algorithmik und Evaluation des Filmmetadaten Such- und Analysesystems libhugin

Bachelorarbeit
an der Hochschule für Angewandte Wissenschaften Hof

Vorgelegt bei:
Prof. Dr. Günther Köhler
Alfons-Goppel-Platz 1
95028 Hof

Vorgelegt von:
Christoph Piechula
Breslaustr. 4
95028 Hof

Hof, 3. August 2014

Abstract

This paper shows and evaluates algorithms used by the previously developed movie metadata search and analysis system *libhugin*. Additionally the behavior of the system under various circumstances is analyzed. Apart that all metadata sources used by *libhugin* are evaluated. The purpose of the analysis of the metadata sources is to evaluate all assumptions previously made. Another purpose of this paper is to discover unknown issues and point up possible solutions.

Danksagung

Mein Dank gilt folgenden Personen: Herrn Prof. Dr. Günther Köhler (Projektbetreuung), Herrn Prof. Dr. Jörg Scheidt (Bereitstellung eines Raumes für den Projektzeitraum), Hans-Joachim Engler, Teegschwendner, Cricetulus, Micrathene Whitneyi, Columbidae und natürlich auch — Alces alces.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

Abkürzung	Bedeutung
API	*Application Programming Interface*
URL	*Uniform Resource Locator*
HTTP	*Hypertext Transfer Protocol*
ID	*Idendifier*
XML	*Extensible Markup Language*
JSON	*JavaScript Object Notation*

1 | Einleitung

1.1 Motivation

Heutzutage wird der Großteil unserer Medien digital konsumiert und verwaltet. Ein großer Teil davon ist, neben der Musiksammlung, die private Filmsammlung. Nach dem Aufzeichnen oder Überspielen von Filmen auf einen Home–Theater–PC [1], müssen die Filmmetadaten, die in der Regel auf der DVD–Hülle stehen, gepflegt werden. Hier bietet oft die Abspielsoftware die Möglichkeit, die Metadaten über diverse Onlinequellen, wie beispielsweise über IMDb (siehe [Link–1]), zu beziehen. Eine andere Möglichkeit, die private Filmsammlung zu pflegen, bieten die sogenannten Movie–Metadaten–Manager. Diese Software ist speziell für das Verwalten von Filmmetadaten konzipiert.

Bei der Pflege der Metadaten ist man jedoch auf die Rahmenbedingungen der jeweiligen Software beschränkt. Nutzt man andererseits mehrere Applikationen um eine Filmsammlung zu pflegen, kommt es oft zu Datenredundanzen, Inkonsistenzen und weiteren Problemen.

Um diese Beschränkungen der aktuellen Applikationen abzumildern, wurde vom Autor das modulare Filmmetadaten Such- und Analysesystem *libhugin* entwickelt [1]. Dieses System ist pluginbasiert und bietet somit dem Benutzer die Möglichkeit, es an die eigenen Bedürfnisse anzupassen.

Diese Arbeit behandelt die Theorie zu dem vom Autor entwickelten System. Des Weiteren sollen die Metadaten der von *libhugin* verwendeten Metadatenquellen untersucht werden, um die bisherigen Annahmen über diese zu bestätigen oder zu revidieren.

1.2 Projektziel

Ziel der Arbeit ist es, die verwendeten Ansätze der Bibliothek zu evaluieren und bisherige Annahmen über Filmmetadaten stichprobenartig anhand der in der Projektarbeit implementierten Metadatenanbieter–Plugins zu untersuchen. Ein Ziel dabei ist, die verwendeten Ansätze nochmals kritisch zu reflektieren und mögliche, bisher noch nicht bekannte, Probleme zu identifizieren.

Neben der Evaluierung der Bibliothek soll die Untersuchung der Metadaten Aufschluß über die weitere Vorgehensweise bei der Weiterentwicklung der Bibliothek geben.

[1] Ein PC-Komponenten basiertes System zum Abspielen von Multimedia–Inhalten

Zusammengefasst sollen der aktuelle Prototyp und die bisherigen Annahmen über die Verteilung und die Probleme mit Metadaten untersucht werden.

1.3 Zielgruppe

Zu der Zielgruppe gehören Entwickler, die an der Weiterentwicklung der Bibliothek beteiligt sind, sowie auch interessierte Personen, die sich einen Überblick über die verwendeten Ansätze von *libhugin* und Unterschiede bei den Metadatenquellen verschaffen wollen.

2 | Probleme bei der Metadatenbeschaffung

Die Metadaten eines Films stehen in der Regel auf der DVD–Hülle oder finden sich in der TV–Programmübersicht. Nach dem Überspielen der eigenen DVD–Sammlung oder dem Aufzeichnen von Sendungen fehlen diese und müssen vom Benutzer nachträglich manuell gepflegt werden.

Die „digitale Filmsammlung" wird in der Regel von sogenannter Home–Theater–Software abgespielt und verwaltet. Hierzu gehören beispielsweise Anwendungen wie das XBMC–Media–Center (siehe [Link–2]) oder Windows–Media–Center (siehe [Link–3]). Diese Software kann in der Regel Metadaten für die digitalisierten Filme beschaffen, ist jedoch oft nur auf bestimmte Onlinequellen beschränkt, die nur eine bestimmte Sprache unterstützen.

Da es für das Speichern der Metadaten keinen durchgesetzten Standard gibt, verwenden die genannten „Media–Center" ein unterschiedliches Format zur Speicherung der Metadaten. Das XBMC–Media–Center verwendet das *nfo*–Format (siehe [Link–4]) und das Windows–Media–Center das *dvdxml*–Format (siehe [Link–5]).

Des Weiteren gibt es sogenannte Movie–Metadaten–Manager–Software, welche primär nur für das Pflegen und Verwalten der digitalen Medien gedacht ist. Zu dieser Art von Software gehört beispielsweise MediaElch (siehe [Link–6]). Die Metadaten–Manager unterstützen oft mehrere Onlinequellen. Die Software erlaubt es häufig, die gepflegten Metadaten zu exportieren, um diese in Kombination mit einer Home–Theater–Abspielsoftware nutzen zu können.

Bei der Pflege von Filmsammlungen von mehreren hundert Filmen kommt es immer wieder zu Problemen. Es gibt hier nicht das Werkzeug der Wahl. Jede Software hat ihre Vor– und Nachteile und die Bedürfnisse der Benutzer sind unterschiedlich.

Zu den generellen Problemen gehören folgende Punkte:

- Filmmetadaten werden nicht gefunden.
- Filmmetadaten sind unvollständig.
- Filmmetadaten sind nur in bestimmter Sprache vorhanden.
- Einsatz von mehreren Onlinequellen schwer oder nicht möglich.

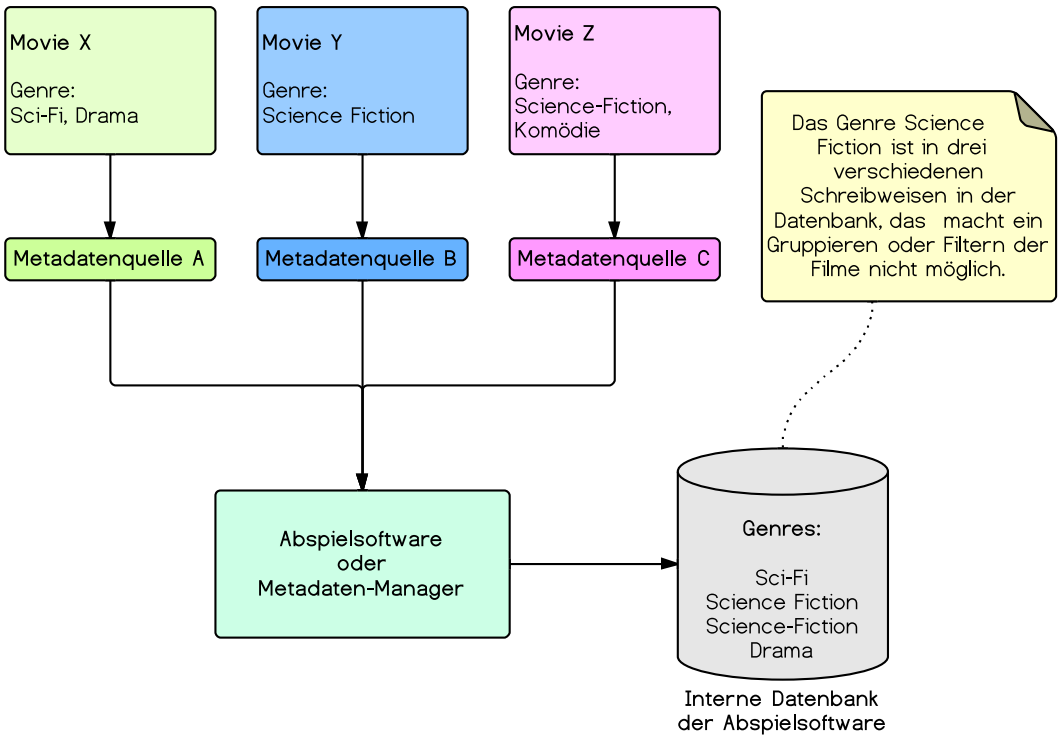

Abbildung 2.1.: Redundante Metadaten beim Bezug von Filmen aus mehreren Onlinequellen.

Je nach Abspielsoftware werden nur bestimmte Onlinequellen für die Beschaffung der Metadaten verwendet. Dies hat zufolge, dass beispielsweise ausländische Filme oder weniger bekannte Filme nicht gefunden werden oder die Inhaltsbeschreibung nur in einer bestimmten Sprache vorliegt oder die Metadaten unvollständig sind.

Unterstützt die Abspielsoftware beziehungsweise der Metadaten–Manager mehrere Onlinequellen, so entstehen häufig aufgrund der nicht normalisierten Metadaten Probleme beim parallelen Bezug der Metadaten aus mehreren Quellen. Das Hauptproblem sind Redundanzen der Metadaten in der internen Datenbank der Abspielsoftware. Diese entstehen hauptsächlich beim Genre, wenn mehrere Filme von unterschiedlichen Onlinequellen bezogen werden.

In Abbildung 2.1 ist das Genre „Science Fiction" bei den drei unterschiedlichen Onlinequellen in einer unterschiedlichen Schreibweise vorhanden. Müssen Metadaten von unterschiedlichen Quellen bezogen werden, weil die Daten für einen bestimmten Film unvollständig oder nicht vorhanden sind, so wird im Beispiel das Genre „Science Fiction" mit drei verschiedenen Schreibweisen in der Datenbank der Abspielsoftware hinterlegt. Dies hat zufolge, dass eine Gruppierung oder Filterung der Filme nach diesem Attribut nicht mehr möglich ist.

Ein weiteres Problem zeigt Abbildung 2.2. Hier ist das Genre–Attribut unterschiedlich detailliert gepflegt.

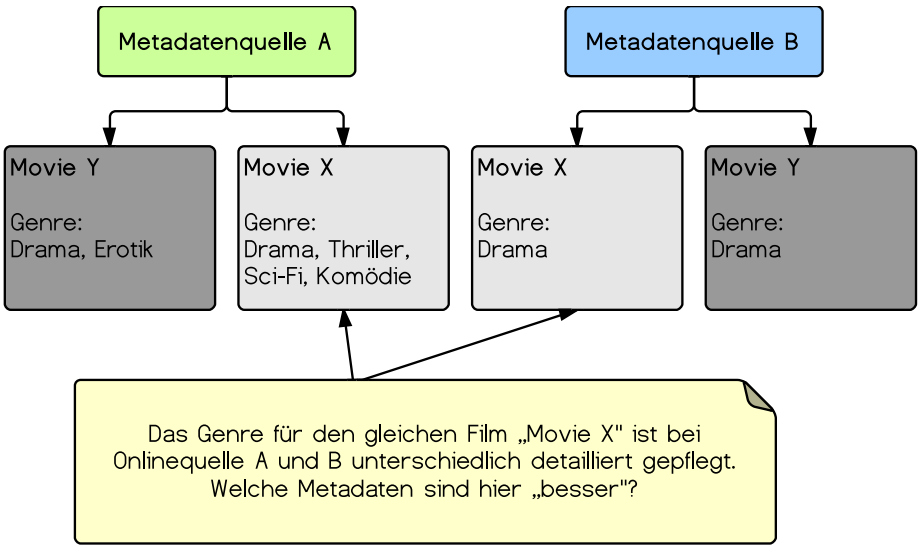

Abbildung 2.2.: Unterschiedlicher Detailgrad im Genre bei verschiedenen Onlinequellen.

Neben den genannten Problemen kommt hinzu, dass unter *unixoiden* Betriebssystemen die Auswahl an gut funktionierenden Filmmetadaten–Managern beschränkt ist, wie in einem Test in der Projektarbeit festgestellt wurde (siehe [1], 3.4.2 Probleme bei Movie–Metadaten–Managern).

Um die aktuell vorhandenen Schwierigkeiten bei der Metadatenpflege zu beheben beziehungsweise abzumildern, wurde das modulare pluginbasierte System *libhugin* entwickelt. Das System fungiert als Bibliothek zur Metadatenbeschaffung und zeigt im Vergleich zu den bestehenden Lösungen eine andere Herangehensweise, die es dem Benutzer erlaubt, das System durch den pluginbasierten Ansatz besser an die eigenen Bedürfnisse anzupassen.

Zusätzlich wurde das System um das Konzept der Metadatenaufbereitung erweitert. Hierdurch soll dem Benutzer die Möglichkeit geboten werden, nachträglich Metadaten automatisiert zu analysieren und Fehler zu bereinigen. Hier wurde ebenso ein pluginbasierter Ansatz gewählt.

Das Hauptaugenmerk von *libhugin* liegt auf der automatisierten Metadatenpflege großer Filmsammlungen von mehreren tausend Filmen.

3 | Übersicht der Software–Architektur

Die Bibliothek wurde in die zwei Teile *libhugin–harvest* (Metadatenbeschaffung) und *libhugin–analyze* (Metadatenaufbereitung) aufgeteilt. Siehe auch Architektur–Übersicht Abbildung 3.1.

Libhugin–harvest

Der *libhugin–harvest* Teil der Bibliothek ist um die folgenden drei Pluginarten erweiterbar:

Provider–Plugins: Diese Plugins sind das „Kernstück" des Projekts und fungieren als „Vermittler" zwischen der Onlinequelle und *libhugin*. Diese Art von Plugin muss von einer Provider–Oberklasse ableiten und die folgenden zwei Methoden implementieren:

- `build_url()`–Methode (baut die URL für den Download zusammen)
- `parse_response()`–Methode (extrahiert die Daten aus der HTTP–Response)

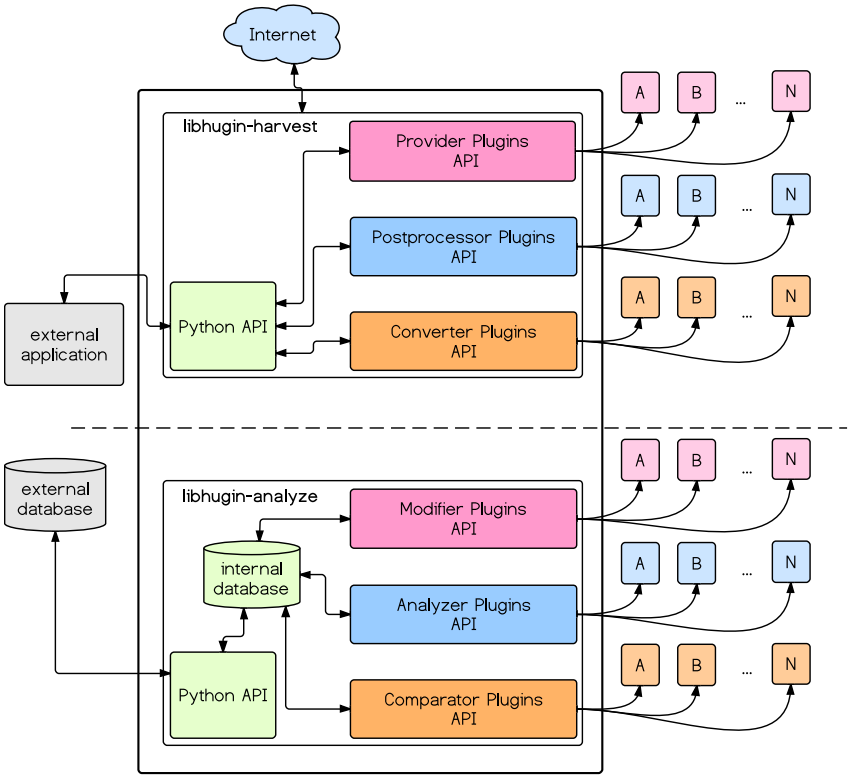

Abbildung 3.1.: Übersicht der Architektur von libhugin.

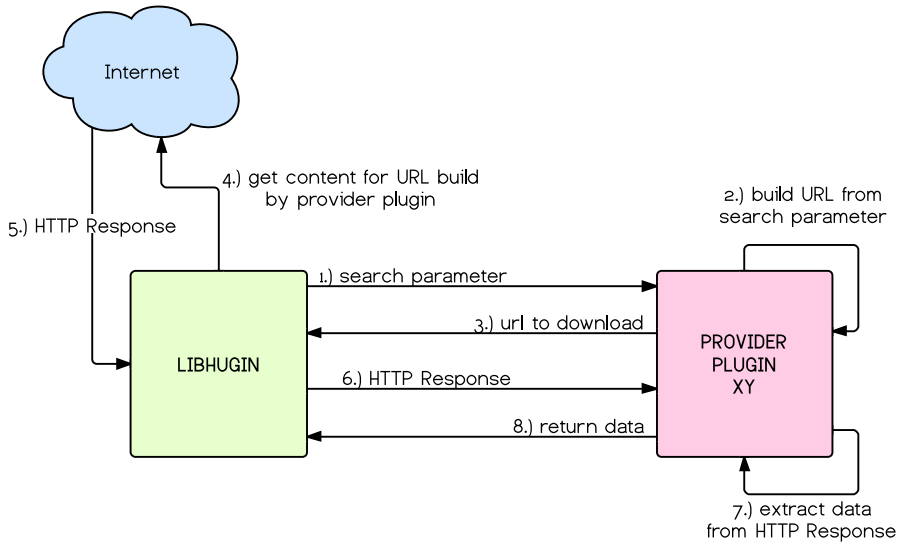

Abbildung 3.2.: Provider-Konzept für die Beschaffung von Metadaten.

Abbildung 3.2 zeigt die grundlegende Funktionsweise. Dabei müssen alle Provider ein vorgegebenes Ergebnisobjekt mit ihren Metadaten befüllen. Für weitere Informationen zum Ergebnisobjekt siehe *libhugin*–API [Link–7].

Postprocessor–Plugins: Diese Plugins sind für das Nachbearbeiten der heruntergeladenen Metadaten zuständig. Diese Plugins müssen eine `process()`–Methode implementieren und von der Postprocessor–Oberklasse ableiten.

Converter–Plugins: Diese Plugins sind für das Exportieren der Metadaten in verschiedene Metadaten–Formate zuständig. Sie müssen von der Converter–Oberklasse ableiten und eine `convert()`–Methode implementieren.

Libhugin–analyze

Der *libhugin-analyze* Teil der Bibliothek dient zur nachträglichen Manipulation und Analyse der Metadaten. Es wird dabei nicht direkt auf den Metadaten gearbeitet, sondern auf einer internen Kopie. Dazu müssen die Metadaten über eine *libhugin–analyze*–Sitzung in die „interne Datenbank" importiert werden. Nachdem die Metadaten analysiert und modifiziert wurden, können diese anschließend wieder ins Produktivsystem zurückgespielt werden.

Hier gibt es die Möglichkeit folgende Pluginarten zu implementieren:

Analyzer–Plugins: Dienen zum Analysieren der Metadaten. Die Plugins müssen von der Analyze–Oberklasse ableiten und eine `analyze()`–Methode implementieren.

Modifier–Plugins: Modifier–Plugins können Metadaten direkt manipulieren. Diese Plugins müssen von der Modifier–Oberklasse ableiten und die `modify()`–Methode implementieren.

Comparator–Plugins: Dieses Plugin–Interface ist experimentell. Es soll zum Vergleich von Filmmetadaten untereinander dienen. Comparator–Plugins müssen von der Comparator–Oberklasse ableiten und eine `compare()`–Methode implementieren.

Weitere Informationen zu der unter Kapitel 2 genannten Problematik oder zum Software–Design selbst werden in der Arbeit zum Projekt *„Design und Implementierung eines modularen Filmmetadaten Such– und Analysesystems"*, siehe [1], sowie in der offiziellen API [Link–8], behandelt.

Für die *libhugin*–Bibliothek wurden in der Projektarbeit die zwei Kommandozeilen Tools Geri (für *libhugin–harvest*) und Freki (für *libhugin-analyze*) entwickelt. Diese Tools demonstrieren die Funktionsweise und Features der Bibliothek und dienen gleichzeitig als einfache Schnittstelle für den direkten Einsatz der Bibliothek. Des Weiteren wurde auch ein konzeptioneller Ansatz für die Integration von *libhugin* in andere Projekte gezeigt. Siehe [1], insbesondere Kapitel 7 Demoanwendungen, für weitere Informationen zu den Funktionen und Features von *libhugin*.

4 | Technische Grundüberlegungen

Bestimmte Teile von *libhugin* arbeiten parallelisiert. Hierzu zählt der Downloadmechanismus, sowie die Möglichkeit einer asynchronen Suchanfrage.

Auf weiteren Einsatz von Parallelisierung wurde verzichtet, da parallele Verarbeitung unter Python aufgrund vom *GIL* (global interpreter lock) nur eingeschränkt möglich ist.

Der *GIL* ist ein Mutex, welcher verhindert, dass mehrere native Threads Python Bytecode gleichzeitig ausführen können. Die Parallelisierung von Funktionen kann zu Performanceeinbußen im Vergleich zur Singlethreaded–Ausführung führen, siehe Abbildung 4.1. Zum Testen wurde das Skript im *Anhang A (GIL Limitierung)* verwendet, welches als Aufgabe die Dekrementierung einer Variablen hat.

Diese Einschränkung gilt jedoch nicht für lange laufende oder blockierende Operationen wie beispielsweise der Zugriff auf die Festplatte (vgl. [2]).

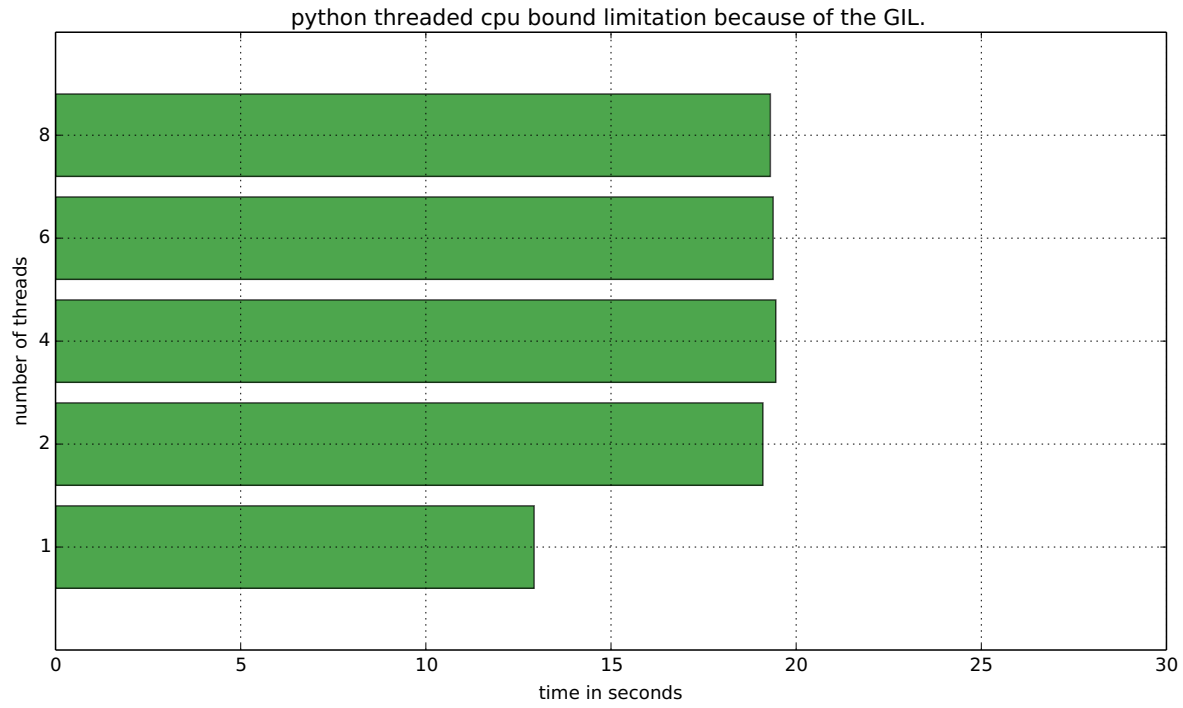

Abbildung 4.1.: Limitierung der Geschwindigkeit durch den global interpreter lock bei CPU-abhängigen Aufgaben. Hier wird über einer Funktion der Wert 100.000.000 dekrementiert.

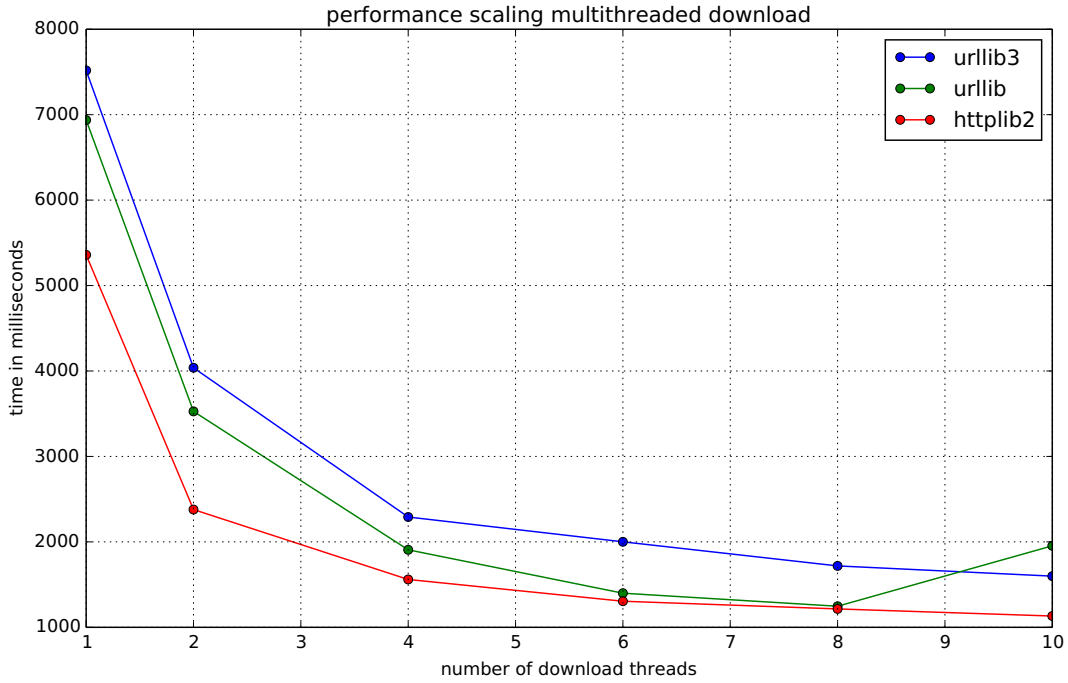

Abbildung 4.2.: Performancevorteil beim Parallelisieren von Downloads. Durchschnitt aus drei Durchläufen, jeweils mit Zugriff auf 15 verschiedene Webseiten.

Da der Zugriff auf Onlinequellen je nach Serverauslastung und Internetanbindung in der Performance stark variiert, wurde das Herunterladen der Metadaten parallelisiert. Das parallele Herunterladen zeigt deutliche Geschwindigkeitsvorteile im Vergleich zur seriellen Verarbeitung (siehe Abbildung 4.2).

Zum Herunterladen wird auf die Python HTTP–Bibliothek *urllib* verzichtet, weil diese grundlegende HTTP–Standards, wie beispielsweise Kompression, nicht unterstützt.

Zwei weitere HTTP–Bibliotheken unter Python sind die beiden freien Implementierungen *urllib3* und *httplib2*, auf welche zurückgegriffen werden kann. Bei aktivierter Kompression, hier ist im RFC1951-RFC1952 der *deflate* und *gzip* Algorithmus vorgesehen, wird der Inhalt vor dem Versenden komprimiert und auf Empfängerseite transparent dekomprimiert. Textdateien lassen sich in der Regel gut komprimieren. Durch die Kompression müssen weniger Daten übertragen werden, was sich bei großen Datenmengen und einer geringen Bandbreite auf die Performance auswirken kann.

Folgende Python–Sitzung zeigt die Standard HTTP–Bibliothek *urllib* der Python Standardbibliothek. Diese erhält den komprimierten Inhalt, kann diesen jedoch nicht dekomprimieren, da dieses HTTP–Standardfeature nicht beherrscht wird:

```
>>> from urllib.request import urlopen
>>> urlopen('http://httpbin.org/gzip').read()
b'\x1f\x8b\x08\x00\xc0\xa5\x8bS\x02\xff5\x8f\xc1n\x830\x10D\xef\xf9\n\xe4s\xec\[...]'
```

Im Gegenzug dazu wird der Zugriff über *urllib3*– und die *httplib2*–Bibliothek auf die gleiche Ressource gezeigt (gekürzte Version):

```
>>> from httplib2 import Http
>>> Http().request('http://httpbin.org/gzip')
b'{\n  "gzipped": true,\n  "headers": {\n    "Accept-Encoding": "gzip, deflate"[...]'

>>> import urllib3
>>> urllib3.PoolManager(1).request(url='http://httpbin.org/gzip', method='GET').data
b'{\n  "gzipped": true,\n  "headers": {\n    "Accept-Encoding": "identity",\n [...]'
```

Aufgrund der genannten Eigenschaften und der vergleichsweise guten Performance (siehe Abbildung 4.2) wurde für *libhugin* die *httplib2*–Bibliothek gewählt. Da diese jedoch nicht Thread–Safe ist, wird hier der in der Google Developer API genannte Ansatz gewählt (siehe [Link–9]), eine Instanz pro Thread zu starten.

Abbildung 4.2 zeigt wie sich das Parallelisieren mehrerer Downloads auf die Performance auswirkt. Hier wurden die drei genannten HTTP–Bibliotheken mit dem Skript in *Anhang B (Httplib Benchmark)* getestet. Der Benchmark wurde mit einer *VDSL* 50Mbit–Leitung durchgeführt.

5 | Algorithmik der Filmsuche

Für die Suche nach Filmmetadaten gibt es unter *libhugin* mehrere Möglichkeiten. Je nach Metadaten–Provider ist eine Suche nach IMDb–ID und Titel möglich. Die IMDb–ID ist eine von IMDb.com festgelegte einzigartige ID für einen Film.

Folgende Python–Shell Sitzung zeigt wie eine Metadaten Suchanfrage funktioniert:

```
>>> from hugin.harvest.session import Session
>>> s = Session()
>>> q = s.create_query(title='The Matrix')
>>> r = s.submit(q)
>>> print(r)
[<tmdbmovie <picture, movie> : The Matrix (1999)>,
 <ofdbmovie <movie> : Matrix (1999)>,
 <filmstartsmovie <movie> : Matrix (1999)>]
```

Beim Erstellen der Sitzung können *libhugin* Konfigurationsparameter übergeben werden, wie beispielsweise:

- Cache Pfad, Pfad zum lokalen HTTP–Anfragen Zwischenspeicher.
- Anzahl paralleler Downloads per Thread

Anschließend muss eine Suchanfrage erstellt werden. Dazu gibt es die Möglichkeit, die Methode `create_query()` zur Hilfe zu nehmen. Hier hat der Benutzer eine Vielzahl von Möglichkeiten, seine Suchanfrage zu konfigurieren.

Der letzte Schritt ist das Absenden der Suchanfrage. Hier gibt es die Möglichkeit einer *synchronen* (`submit()`–Methode) oder einer *asynchronen* Anfrage (`submit_async()`–Methode). Der Hauptunterschied ist, dass die *asynchrone* Anfrage im Gegensatz zu der *synchronen* nicht blockiert. Der Aufrufer der Methode kann also in der Zwischenzeit andere Aufgaben erledigen.

Siehe [1] und *libhugin* API [Link–8] für eine vollständige Liste der Konfigurationsparameter der Session und der Query.

5.1 Standardsuche

Bei der Suchanfrage über den Filmtitel wird von den Onlinequellen in der Regel eine Liste mit mehreren Möglichkeiten geliefert. Das Provider–Plugin muss anschließend die Filmtitel mit der größten Übereinstimmung herausfinden. Für die Ähnlichkeit bei der Suche nach übereinstimmenden Zeichenketten wurde ein Ähnlichkeitsmaß definiert, welches eine Spanne von 0.0 (keine Ähnlichkeit) bis 1.0 (volle Übereinstimmung) aufweist.

Der Vergleich der Zeichenketten sollte möglichst fehlertolerant sein und Zeichenketten mit der höchsten Übereinstimmung liefern.

Ein simpler Vergleich wie beispielsweise

```
>>> "The Matrix" == "The Matrix"
True
>>> "The Matrix" == "The matrix"
False
```

funktioniert nur bei exakt den gleichen Zeichenketten. Des Weiteren ist so auch die Umsetzung einer Werte–Spanne nicht möglich. Für den Vergleich von Zeichenketten bietet die Python Standard–Bibliothek das *difflib*–Modul. Das Modul erlaubt es, zwei Sequenzen zu vergleichen. Es arbeitet mit dem Ratcliff–Obershelp–Algorithmus und hat eine Komplexität von $O(n^3)$ im *worst case* und eine erwartete Komplexität von $O(n^2)$. Der Algorithmus basiert auf der Idee, die Anzahl der übereinstimmenden Sequenzen (in beiden Zeichenketten übereinstimmende Folgen von einem oder mehreren Zeichen) zu zählen. Für weitere Details zum Algorithmus, siehe [3], [4].

Ein weiteres Maß für die Ähnlichkeit von Zeichenketten ist die Hamming–Distanz. Diese Distanz arbeitet nach der Idee, die „Ersetzungen" zu zählen. Der Algorithmus hat jedoch die Einschränkung, dass er sich nur auf gleich lange Zeichenketten anwenden lässt (vgl. [4], [5]).

Ein weiterer Algorithmus, der für Zeichenkettenvergleiche eingesetzt wird, ist der Levenshtein–Algorithmus (auch Levenshtein–Distanz genannt). Der Algorithmus hat eine Laufzeitkomplexität von $O(nm)$, n und m repräsentieren jeweils die Längen der Zeichenketten. Die Levenshtein–Distanz basiert auf der Idee, die minimalen Editiervorgänge (Einfügen, Löschen, Ersetzen), um von einer Zeichenkette auf eine andere zu kommen (vgl [4], [5], [6] und [7]), zu zählen. Die normalisierte Levenshtein–Distanz bewegt sich zwischen 0.0 (Übereinstimmung) und 1.0 (keine Ähnlichkeit).

Eine Erweiterung der Levenshtein–Distanz ist die Damerau–Levenshtein–Distanz. Diese wurde um die Funktionalität erweitert, vertauschte Zeichen zu erkennen. Um die Zeichenkette *„The Matrix"* nach *„Teh Matrix"* zu überführen, sind bei der Levenshtein–Distanz zwei Operationen nötig, die Damerau–Levenshtein–Distanz hingegen benötigt nur eine Operation wie die folgende *IPython*–Sitzung zeigt:

```
>>> from pyxdameraulevenshtein import damerau_levenshtein_distance
>>> from distance import levenshtein as levenshtein_distance
>>> levenshtein_distance("the matrix", "teh matrix")
2
>>> damerau_levenshtein_distance("the matrix", "teh matrix")
1
```

Von der Levenshtein- und Damerau–Levenshtein–Distanz gibt es jeweils eine normalisierte Variante. Hierbei bewegt sich die Distanz zwischen 0.0 und 1.0. Dies wird dadurch erreicht, indem die Anzahl der Operationen durch die Länge der längeren der beiden Zeichenketten geteilt wird.

Da es bei der Filmsuche zu vielen Zeichenkettenvergleichen kommt sollte der Algorithmus zum Vergleich von Zeichenketten performant sein.

Um die jeweiligen Algorithmen beziehungsweise ihre Implementierungen bezüglich der Performance zu überprüfen, wurde eine Messung mit den folgenden unter Python verfügbaren Implementierungen durchgeführt:

- difflib, Modul aus der Python–Standardbibliothek (Ratcliff-Obershelp)
- pyxDamerauLevenshtein, auf C basierte Implementierung von Damerau–Levenshtein
- distance, externes Modul mit Levenshtein–Implementierung in C

Abbildung 5.1 zeigt, dass die Laufzeit–Komplexität bei allen drei Algorithmen ähnlich ist. Des Weiteren zeigt die Abbildung, dass die beiden Implementierungen *distance* (C) und *pyxDamerauLevenshtein* (C) sehr performant im Vergleich zur *difflib* (Python) Implementierung arbeiten. Aufgrund der Tatsache, dass der Damerau–Levenshtein–Algorithmus vertauschte Zeichen „erkennen" kann und gleichzeitig performant implementiert ist, wurde er für den Einsatz in der Bibliothek gewählt.

Der Benchmark wurde mit dem Skript aus *Anhang C (Analyse Zeichenkettenvergleich)* durchgeführt.

Je nach verwendeten Algorithmus variiert das Ergebnis leicht. Das liegt daran, dass die Algorithmen eine unterschiedliche Idee verfolgen.

Beim Levenshtein–Algorithmus wird eine Distanz (0.0 volle Übereinstimmung, 1.0 keine Übereinstimmung) zum Ermitteln der Ähnlichkeit zweier Zeichenketten angewandt. Beim Ratcliff–Obershelp–Algorithmus hingegen wird die Ähnlichkeit durch ein Ähnlichkeitsmaß (0.0 keine Übereinstimmung, 1.0 volle Übereinstimmung) ermittelt. Um eine Vergleichbarkeit des Ergebnisverhaltens der beiden Algorithmen herzustellen, wird die vom Levenshtein–Algorithmus errechnete Distanz von Eins subtrahiert. So lässt sich das Verhalten der beiden Algorithmen besser miteinander vergleichen.

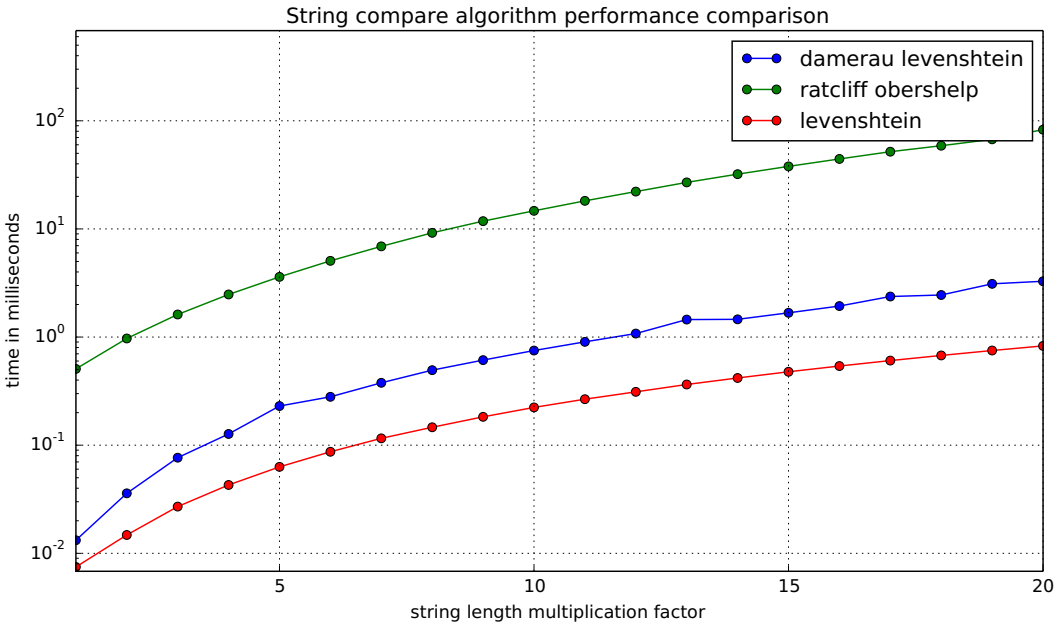

Abbildung 5.1.: Performancevergleich der Algorithmen für den Zeichenkettenvergleich. Verglichen werden jeweils die beiden Basis-Zeichenketten 'Erdmännchen' und 'Khaleesi' miteinander, der string length multiplication factor gibt indirekt die Länge der jeweiligen Zeichenkette an. Ein Faktor von beispielsweise zwei bedeutet, dass an dieser Stelle die beiden Zeichenketten "ErdmännchenErdmännchen" und "KhaleesiKhaleesi" miteinander verglichen werden.

Folgende interaktive *IPython*–Sitzung zeigt das Ergebnisverhalten von *difflib* und *pyxDamerauLevenshtein*.

```
>>> difflib.SequenceMatcher(None, "Katze", "Fratze").ratio()
0.7272727272727273
>>> 1 - normalized_damerau_levenshtein_distance("Katze", "Fratze")
0.6666666666666667
```

Weitere Werte, um die unterschiedliche Wertung der beiden Algorithmen zu demonstrieren, finden sich in der Tabelle 5.2 und 5.3. Die Werte wurden mit dem Skript in *Anhang D (Rating Differenz)* ermittelt.

	Superman	**Batman**	**Iron-Man**	**Spiderman**
Superman	1.0	0.38	0.25	0.67
Batman	×	1.0	0.25	0.33
Iron-Man	×	×	1.0	0.22
Spiderman	×	×	×	1.0

Abbildung 5.2.: Ähnlichkeitswerte ermittelt mit Damerau-Levenshtein.

Da der Vergleich von der Groß– und Kleinschreibung abhängig ist, fällt die Ähnlichkeit der Titel „*Sin*" und „*sin*", wie folgende *IPython*–Sitzung zeigt, unterschiedlich aus:

	Superman	Batman	Iron-Man	Spiderman
Superman	1.0	0.43	0.38	0.82
Batman	×	1.0	0.29	0.4
Iron-Man	×	×	1.0	0.35
Spiderman	×	×	×	1.0

Abbildung 5.3.: Ähnlichkeitswerte ermittelt mit Ratcliff-Obershelp.

```
>>> 1 - normalized_damerau_levenshtein_distance("sin", "Sin")
0.6666666666666667
```

Um dieses Problem zu beheben, wird die gesuchte Zeichenkette vor dem Vergleich normalisiert. Dies geschieht indem alle Zeichen der Zeichenkette in Klein– beziehungsweise alternativ in Großbuchstaben umgewandelt werden. Folgendes Beispiel zeigt die Normalisierung mittels der in Python integrierten `lower()`–Funktion:

```
>>> 1 - normalized_damerau_levenshtein_distance("sin".lower(), "Sin".lower())
1.0
```

Während der Entwicklung ist aufgefallen, dass der implementierte OFDb–Provider den Film *„The East (2013)"* nicht finden konnte. Nach längerer Recherche und Ausweitung der gewünschten Ergebnisanzahl auf 100 Ergebnisse, wurde festgestellt, dass der Film auf dem letzten Platz der Suchergebnisse (Platz 48) zu finden war. Die vorherigen Plätze waren mit Filmtiteln wie „The Queen of the East" oder „Horror in the East" besetzt.

Dies lag daran, dass der Film auf dieser Online–Plattform in der Schreibweise *„East, The"* gepflegt ist. Dies ist eine valide und nicht unübliche Schreibweise, um Filme alphabetisch schneller zu finden.

Betrachtet man die Ähnlichkeit der beiden Zeichenketten, so stellt man fest, dass bei dieser Schreibweise, je nach Algorithmus, eine geringe bis gar keine Ähnlichkeit vorhanden ist, wie folgende *IPython* Sitzung zeigt:

```
>>> import difflib
>>> from pyxdameraulevenshtein import normalized_damerau_levenshtein_distance
>>> difflib.SequenceMatcher(None, "The East", "East, The").ratio()
0.47058823529411764
>>> 1 - normalized_damerau_levenshtein_distance("The East", "East, The")
0.0
```

Um dieses Problem zu umgehen, müssen die Filmtitel auf ein bestimmtes Schema normalisiert werden. Ein möglicher Ansatz wäre, den Artikel zu entfernen. Dies würde jedoch das Problem mit sich bringen, dass Filme wie *„Drive (2011)"* und *"The Drive (1996)"* fälschlicherweise als identisch erkannt werden würden. Ein weiteres Problem besteht darin, dass der Artikel–Ansatz sprachabhängig ist.

Die Satztrennungszeichen zu entfernen und die einzelnen Wörter des Titels alphabetisch zu sortieren ist ein anderer Ansatz, der bei *libhugin* gewählt wurde.

Anhand des Beispieltitel „*East, The*" wird folgend das Vorgehen erläutert:

1. Titel auf Kleinschreibung umwandeln → `'east, the'`
2. Satztrennungszeichen wie „,", „-" und „:" werden entfernt → `'east the'`
3. Titel anhand der Leerzeichen aufbrechen und in Liste umwandeln → `['east', 'the']`
4. Liste alphabetisch sortieren und in Zeichenkette zurückwandeln → `'east the'`
5. Vergleich mittels Damerau–Levenshtein Algorithmus

Wendet man diesen Ansatz auf „The East" und „East, The" an, so erhält man in beiden Fällen die Zeichenkette "east the". Die Umsetzung dieses Algorithmus bei der Titelsuche löst das Problem beim OFDb–Provider. Der eben genannte Film wird durch die Normalisierung gefunden und erscheint an der ersten Position.

Diese Vorgehensweise normalisiert ebenso die Personensuche. Hier wird beispielsweise der Name „*Emma Stone*" und „*Stone, Emma*" in beiden Fällen zu der Zeichenkette `'emma stone'`.

Abbildung 5.4 zeigt wie sich die im Kapitel 5.1 Standardsuche vorgenommenen Anpassungen auf die Performance auswirken. Wie in der Auswertung zu sehen ist, fallen die Anpassungen kaum ins Gewicht. Das Laufzeitverhalten hat sich nicht verschlechtert, anfangs sind lediglich kleine Performanceeinbußen messbar, bei längeren Zeichenketten ab ungefähr 20 Zeichen ist kein Unterschied messbar. Aufgrund dieser Tatsache kann der Algorithmus trotz Anpassungen in *libhugin* verwendet werden, ohne dass man mit Performanceeinbußen rechnen muss.

Ein weiteres Attribut, das bei der Suche von Filmen angegeben werden kann, ist das Erscheinungsjahr. Dieses wird verwendet, um Suchergebnisse genauer einzugrenzen.

Wird der Titel und ein Erscheinungsjahr bei der Suche angegeben, so kann der „richtigere" Film näherungsweise durch das Erscheinungsjahr ermittelt werden. Beim simplen Vergleich des Jahres mittels Damerau–Levenshtein Algorithmus ergibt sich hier jedoch ein neues Problem.

Bei zusätzlicher Anwendung des Damerau–Levenshtein–Algorithmus auf das Erscheinungsjahr kann es zu dem Fall kommen, dass das logisch gesehen „nähere" Erscheinungsjahr als „schlechter" gewertet wird. Das liegt daran, dass es Fälle gibt, bei denen der logische Jahresunterschied zum Suchstring geringer sein kann, als der Zeichenkettenunterschied. In diesem Fall würde ein Film, der den gleichen Titel hat, aber zeitlich gesehen viel weiter vom gesuchten Film entfernt ist, als „besser" bewertet werden.

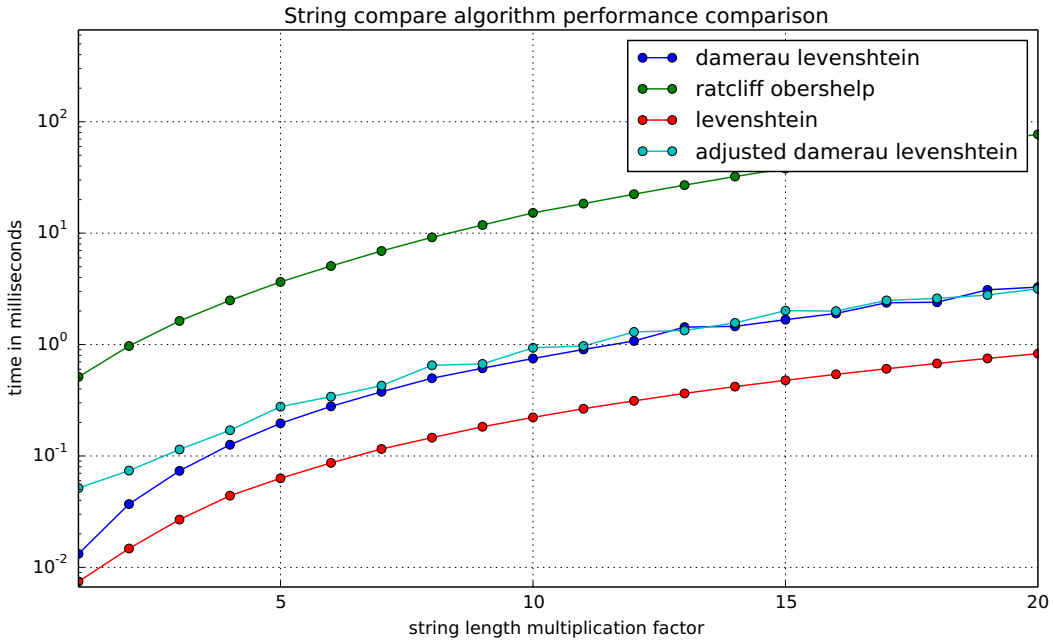

Abbildung 5.4.: Angepasster Algorithmus auf Basis von Damerau-Levenshtein im Vergleich zu den ursprünglichen Algorithmen aus Abbildung 5.1. Verglichen werden jeweils die beiden Basis-Zeichenketten 'Erdmännchen' und 'Khaleesi' miteinander, der string length multiplication factor gibt indirekt die Länge der jeweiligen Zeichenkette an. Ein Faktor von beispielsweise zwei bedeutet, dass an dieser Stelle die beiden Zeichenketten "ErdmännchenErdmännchen" und "KhaleesiKhaleesi" miteinander verglichen werden.

Folgende *IPython*–Sitzung zeigt die Problematik:

```
>>> 1 - normalized_damerau_levenshtein_distance("Drive 2000", "Drive 2011")
0.8
>>> 1 - normalized_damerau_levenshtein_distance("Drive 2000", "Drive 1997")
0.6
```

Bei separater Betrachtung der Zeichenkette für das Jahr würde die Differenz noch größer ausfallen, da die beiden Zeichenketten „1997" und „2000" keine Ähnlichkeit aufweisen, die Zeichenketten „2000" und „2011" eine Ähnlichkeit von 0.5 aufweisen.

Logisch betrachtet ist das Jahr „1997" jedoch viel näher an dem gesuchten Erscheinungsjahr. Was im Beispiel darauf hindeuten würde, dass der Benutzer das exakte Jahr nicht mehr wusste, jedoch den Zeitraum mit einer Abweichung von drei Jahren angeben konnte.

Die genannte Problematik äußert sich beispielsweise auch bei Film–Remakes oder Filmen, die mit einer Ungenauigkeit von ± 1 Jahr auf einer Plattform eingepflegt wurden. Nach Beobachtung des Autors gibt es hier zwischen den Onlinequellen für den gleichen Film vereinzelt Differenzen beim Erscheinungsjahr.

Ob dieser Umstand weiterhin präsent ist, beziehungsweise wie oft dieser Fall vorkommt, zeigt die Auswertung der Stichprobe der Metadaten mehrerer Onlinequellen (siehe Analyse der Erscheinungsjahrdifferenz 7.3).

Die gerade im Kapitel 5.1 Standardsuche erläuterten Probleme beim Titelvergleich mit Erscheinungsjahr können durch das separate Betrachten des Erscheinungsjahres und das Einführen einer Gewichtung abgemildert werden.

Hierzu wird zuerst mittels folgender Formel die Ähnlichkeit für das Erscheinungsjahr ermittelt:

$$year\_similarity(year_a, year_b, max_{years}) = 1 - min\left\{1, \frac{|year_a - year_b|}{max_{years}}\right\}$$

max_{years} ist hierbei die maximale Anzahl von Jahren (Jahresdifferenz), die betrachtet werden sollen.

Anschließend wird das Jahr noch zusätzlich gewichtet, da der Titel wichtiger als das Erscheinungsjahr ist. Durch die Gewichtung soll die Wichtigkeit des Titels zusätzlich sichergestellt werden.

Die Berechnung der Ähnlichkeit samt Gewichtung kann anschließend mit folgender Formel berechnet werden:

$$similarity(t_a, y_a, t_b, y_b) = \frac{string\_similarity\_ratio(t_a, t_b) \times weight + year\_similarity(y_a, y_b)}{weight + 1}$$

t_a, t_b sind die jeweiligen Titel.

y_a, y_b sind die jeweiligen Erscheinungsjahre.

string_similarity_ratio() ist die angepasste Damerau–Levenshtein Funktion für den Zeichenkettenvergleich.

$weight$ ist hierbei der Gewichtungsfaktor für den Titel. Eine Gewichtung von beispielsweise fünf lässt den Titel fünf mal stärker als das Jahr in das Ergebnis einfließen. Je höher der Gewichtungsfaktor $weight$, desto stärker fließt der Titel ins Ergebnis ein.

Durch die Gewichtung des Titels fällt ein falsch gepflegtes Erscheinungsjahr nicht so stark ins Gewicht wie ein „Buchstabendreher" beim Titel. Dies ist ein gewolltes Verhalten, da das Jahr nur unterstützend beim Filtern der Ergebnismenge verwendet werden soll.

Titel	Rating mit Gewichtung, weight=3	Rating ohne Gewichtung
Matrix 1999	1.0	1.0
Matrix 2000	0.983	0.636
Matrix 1997	0.967	0.909
Matrix 2001	0.967	0.636
Matrix, The 1999	0.7	0.538
The Matrix 2013	0.467	0.467
The East 1999	0.438	0.538

Abbildung 5.5.: Unterschied im Rating bei gewichteter Betrachtung des Titels. Gewichtetes Rating mit angepasstem Damerau-Levenshtein-Algorithmus, nicht gewichtetes Rating mit Standard Damerau-Levenshtein-Algorithmus. Alle in der Tabelle genannten Titel wurden jeweils mit der Zeichenkette „Matrix 1999" verglichen.

Abbildung 5.5 zeigt das Rating mit und ohne Gewichtung für die Zeichenkette „Matrix 1999". Hier wurde für die Gewichtung exemplarisch der Wert $n = 3$ gewählt. Das Skript für die Auswertung findet sich im *Anhang E (Gewichtetes Rating)*.

5.2 IMDb–ID Suche

Ob die Suche nach der IMDb–ID möglich ist, hängt von der jeweiligen Onlinequelle ab. Onlinequellen wie TMDb, OFDb oder auch OMDb unterstützen direkt die Suche über die IMDb–ID. Andere Onlinequellen, wie das Filmstarts- oder das Videobuster–Portal unterstützen keine Suche über IMDb–ID.

Um trotzdem eine onlinequellenübergreifende Suche über die IMDb–ID zu ermöglichen, bietet die *libhugin–harvest*–Bibliothek den sogenannten „Lookup–Mode".

Hierbei wird intern vor der Metadatensuche ein sogenannter *Lookup* durchgeführt, um zu der gesuchten IMDb–ID den passenden Filmtitel zu ermitteln. Dies ist über die Suche auf IMDb.com möglich. Die Filme auf der Seite sind jeweils unter der jeweiligen IMDb–ID eingepflegt. Die URL für den Film *„Only god forgives (2013)"* mit der IMDb–ID `tt1602613` ist wie folgt aufgebaut:

- http://www.imdb.com/title/tt1602613

Wenn der *Lookup–Mode* aktiviert wird, wird vor der Kommunikation mit den Provider–Plugins ein *Lookup* über `http://imdb.com` getriggert. Hierbei wird die URL aus der zu suchenden ID zusammengesetzt und eine IMDb Anfrage gestartet. Anschließend wird auf dem zurückgelieferten HTTP–Response ein regulärer Ausdruck ausgeführt, welcher die Zeichenkette bestehend aus `<Titelname> <(4-stellige Jahreszahl)>` extrahiert.

Der algorithmische Ansatz sieht unter Python wie folgt aus:

```
>>> imdbid = "tt1602613"  # id for only god forgives
>>> request = requests.get('http://www.imdb.com/title/{}'.format(imdbid))
>>> title, year = re.search('\>(.+?)\s*\((\d{4})', request.text).groups()
>>> print(title, year)
'Only God Forgives 2013'
```

Nach dem Extrahieren der Attribute Titel und Erscheinungsjahr wird die Query mit den Suchparametern, welche an alle Provider–Plugins für die Suche weitergegeben werden, mit diesen ergänzt. Die Provider–Plugins, die keine IMDb–ID unterstützen, können so eine Suche über den Titel und das Erscheinungsjahr durchführen. Für den Benutzer sieht dies nach außen so aus, als würde jeder Provider eine IMDb–ID Suche unterstützen.

5.3 Unschärfesuche

Die Onlinequellen der implementierten Provider, TMDb, IMDb, OFDb, OMDb, Filmstarts und Videobuster benötigen in der Regel exakte Suchanfragen. Bei einem Tippfehler wie „*Unly god forgives*" (Originaltitel: „*Only god forgives*"), wird der Film von den genannten Online–Plattformen nicht gefunden.

```
>>> from hugin.harvest.session import Session
>>> s = Session()
>>> q = s.create_query(title='Unly god forgives', fuzzysearch=False)
>>> r = s.submit(q)
>>> print(r)
[]
```

Diesen Fehler auf Seite von *libhugin* zu beheben ist schwierig. Man müsste eine große Datenbank an Filmtiteln pflegen und aktuell halten und könnte so mit Hilfe dieser den Fehler des Benutzers korrigieren, indem man die ähnlichste aller Zeichenketten aus der Datenbank nehmen würde. Mit der angepassten Damerau–Levenshtein–Ähnlichkeit, die *libhugin* zum Zeichenkettenvergleich anbietet, hätte die falsche Anfrage eine Ähnlichkeit von 0.94.

Eine lokale beziehungsweise zentrale Datenbank aufzubauen wäre möglich, da die Informationen beziehungsweise Metadaten online auf vielen Plattformen verfügbar sind. Diese Datenbank aktuell zu halten ist jedoch schwierig, da nicht bekannt ist auf welchen Plattformen ein Film überhaupt gepflegt ist, beziehungsweise wie aktuell die gepflegten Informationen sind.

Um dieses Problem trotz der genannten Schwierigkeiten zu lösen, bedient sich *libhugin* eines anderen Ansatzes. *Libhugin* delegiert die Information, wie es ein Mensch auch machen würde, an eine Suchmaschine. Im konkreten Fall wird hierbei ein *Lookup* über die Suchmaschine von Google getriggert.

Über die *„I'm Feeling Lucky"*–Funktionalität erlaubt es Google über Parameter die Suchanfrage so zu konfigurieren, dass als Antwort keine Liste mit Suchergebnissen zurückgeliefert wird, sondern die Seite mit der höchsten Übereinstimmung zum Suchergebnis. Hierzu muss die Suchanfrage die Option `btnI=1` als URL–Queryparameter enthalten. Folgendes Beispiel zeigt die Suchanfrage zum Wikipedia–Artikel „Hauskatze" mit Parameter für die *„I'm Feeling Lucky"*–Funktionalität:

- http://www.google.com/search?hl=de&q=Hauskatze&btnI=1

Gibt man diese URL im Browser ein, so wird beispielsweise direkt der Wikipedia–Artikel zur Hauskatze [1] angezeigt und nicht die Seite der Google–Suchanfrage wie es in der Regel der Fall wäre.

Libhugin bedient sich dieser Funktionalität und führt einen *Lookup* mit den Parametern *Filmtitel*, *Erscheinungsjahr*, *imdb* und *movie* aus. Anschließend wird die zurückgegebene URL betrachtet und aus dieser die IMDb–ID extrahiert.

Folgende *IPython*–Sitzung zeigt den Ansatz:

```
>>> fmt = 'http://www.google.com/search?hl=de&q={title}+{year}+imdb+movie&btnI=1'
>>> url = requests.get(fmt.format(title='Drive', year='2011'))).url
>>> imdbid = re.findall('\/tt\d*/', url)
>>> imdbid.pop().strip('/')
'tt0780504'
```

Hier wurde der Ansatz gewählt, die IMDb–ID aus der URL mit einem regulären Ausdruck zu parsen. Dies erspart das Parsen der kompletten HTTP–Response, was deutlich aufwendiger wäre.

Dies geschieht vor der Kommunikation mit den Provider–Plugins. Anschließend wird die Suche mit der IMDb–ID normal fortgesetzt. Alternativ wäre hier der Ansatz über den Filmtitel, wie beim IMDb–ID–zu–Titel–*Lookup* möglich. Diese Funktionalität lässt sich durch das zusätzliche Aktivieren des „IMDb–Lookup"–Mode realisieren.

5.4 Normalisierung des Genre

Die Normalisierung der Metadaten aus unterschiedlichen Quellen ist schwierig, da es bei den Filmmetadaten keinen einheitlichen Standard gibt. Um fehlerhafte oder fehlende Metadaten über unterschiedliche Quellen zu ergänzen, müssen die Metadatenattribute, insbesondere das Genre, aufgrund der in Kapitel 2 gelisteten Problematik normalisiert werden.

Durch den in Kapitel 2 (siehe Abbildung 2.1, Abbildung 2.2) genannten Umstand werden die Genreinformation redundant in der Datenbank der Abspielsoftware abgelegt, wie beispielsweise dem

[1] http://de.wikipedia.org/wiki/Hauskatze

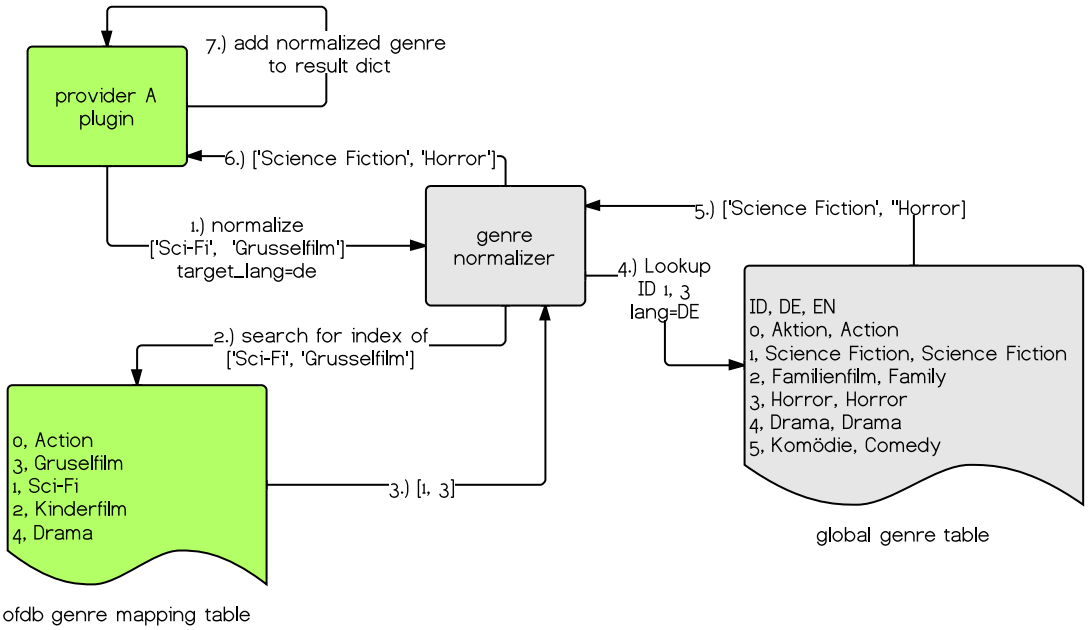

Abbildung 5.6.: Normalisierung der Genreinformationen anhand statischer Mapping-Tabellen.

XBMC–Media–Center. Es ist nicht mehr möglich, ein Filmgenre eindeutig zu identifizieren. Es ist somit weder eine Gruppierung nach diesem Genre noch eine eindeutige Filterung möglich.

Dieses Problem betrifft grundsätzlich alle Filmmetadaten–Attribute, jedoch lassen sich andere Attribute wie die Inhaltsbeschreibung problemlos austauschen, da diese von Natur aus individuell ist und sich somit nicht normalisieren lässt.

Da das Filmgenre, neben der Inhaltsbeschreibung und Filmbewertung, nach Meinung des Autors, zu den wichtigsten Auswahlkriterien bei Filmen zählt, wurde bei *libhugin* ein statisches Konzept der Normalisierung umgesetzt.

Die Normalisierung bei *libhugin* bildet hierzu jedes Genre einer Onlinequelle auf einem globalen Genre ab. Die Normalisierung erfolgt über eine statische Genre–Tabelle, welche der Autor eines Provider–Plugins bereitstellen muss. Der Nachteil dieser Variante ist, dass das Genrespektrum der Onlinequelle bekannt sein muss. Das Provider–Genre wird über einen Index auf einem globalen Genre abgebildet. Abbildung 5.6 zeigt konzeptuell die Vorgehensweise beim „Normalisieren" der Genreinformationen.

Wird keine „Genremapping–Tabelle" bereitgestellt, so kann das Genre nicht normalisiert werden. In diesem Fall kann es zu der oben genannten Problematik kommen. Das Genremapping muss pro Sprache gepflegt werden, der Prototyp besitzt im aktuellen Zustand eine globale Genre–Tabelle für die deutsche und die englische Sprache.

Ein weiterer Ansatz bei der Genrenormalisierung war die automatische Erkennung des Genres anhand der Wortähnlichkeit. Dies erwies sich jedoch als nicht praxistauglich. Eine automatische Genreerkennung benötigt einen Wortschatz aus Referenz–Genres, mit welchen das unbekannte Provider–Genre verglichen werden muss. Bei Genres wie Science–Fiction, Drama oder Thriller funktioniert das System noch relativ gut. Kommen aber seltene oder unbekannte Genrenamen wie „Mondo" oder „Suspense" hinzu, kann je nach Referenz–Wortschatz keine Übereinstimmung mehr erfolgen. Hier wäre noch ein semiautomatischer Ansatz denkbar, welcher automatisiert Genres erkennt und im Fall eines unbekannten Genre dieses in eine Liste aus nicht zugeordneten Genres hinzufügt, welche dann vom Benutzer korrigiert werden können. Dies ist jedoch bei einer Software–Bibliothek wie sie durch *libhugin* bereitgestellt wird, weniger praktikabel.

Ein weiteres Problem, das hier jedoch hinzukommt, besteht darin, dass das Genre an sich in keiner Form standardisiert ist. Je nach Onlinequelle gibt es Genrebezeichnungen wie Animationsfilm oder Kinderfilm, welche jedoch im engeren Sinne nicht zum „Filmgenre"–Begriff gezählt werden dürften (siehe [Link–10]). Des Weiteren kommt hinzu, dass im Laufe der Zeit immer wieder neue Genres entstanden sind.

5.5 Suchstrategien

Der Prototyp der *libhugin–harvest*–Bibliothek unterstützt zwei verschiedene Suchstrategien. Eine *„deep"*–Strategie und eine *„flat"*–Strategie. Diese beiden Strategien sollen dem Benutzer die Kontrolle über die „Suchtrefferart" geben.

Jedes Provider–Plugin hat aktuell eine vergebene Priorität. Diese ist im Prototypen von *libhugin* manuell vergeben worden. Die Priorität ist ein Integer–Wert im Bereich `0-100`. Je höher die Priorität, desto mehr wird ein Provider beim abschließenden Filtern der Ergebnisse berücksichtigt.

Die gefundenen Ergebnisse können einerseits nach Provider–Priorität betrachtet oder aber nach „Ergebnisqualität" betrachtet werden. Aus diesem Grund wurde die *„deep"*– und die *„flat"*–Suchstrategie implementiert.

Bei der *„deep"*–Strategie werden die Ergebnisobjekte nach Provider (Priorität) gruppiert und die Ergebnisse innerhalb jeder Gruppe nach Übereinstimmung mit der gesuchten Zeichenkette sortiert.

Anschließend werden die Ergebnisse, angefangen beim Provider mit der höchsten Priorität, zurückgeliefert bis die gewünschte Anzahl an Ergebnissen zurückgegeben wurde (siehe Abbildung 5.7).

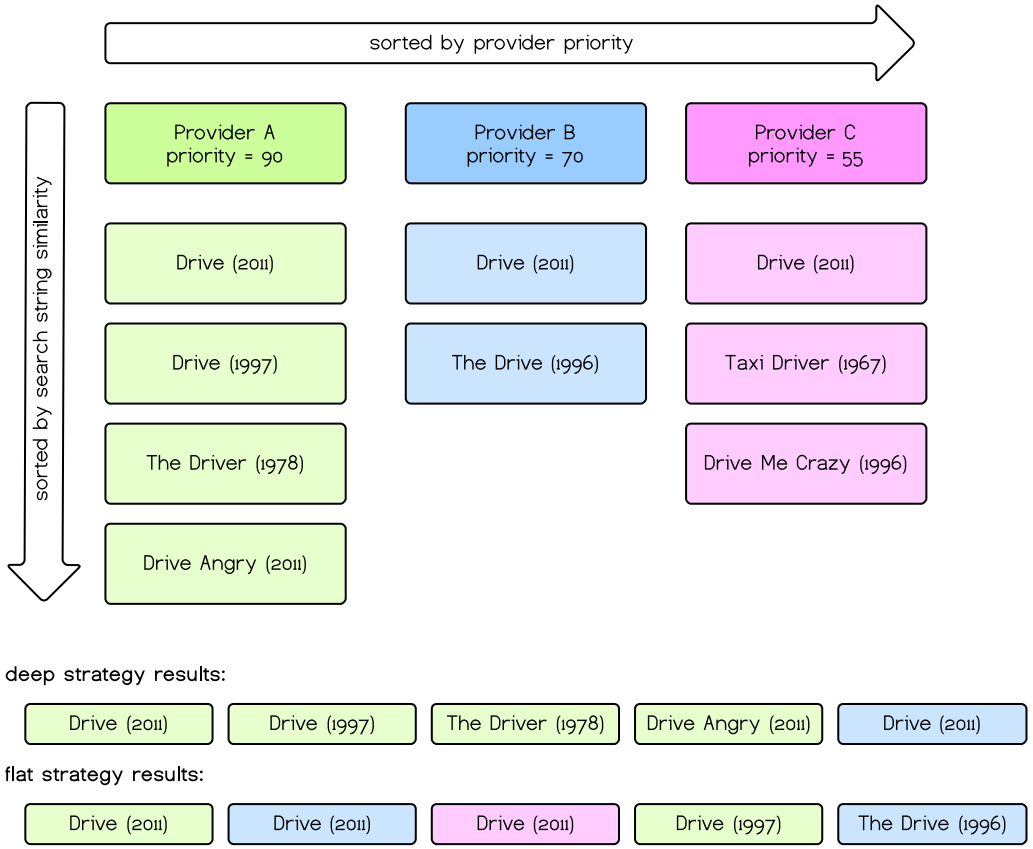

Abbildung 5.7.: Suchstrategien. Suche nach dem Film „Drive (2011)" mit der Begrenzung der Suchergebnisse auf fünf.

Das folgende Beispiel zeigt das tatsächliche Ergebnis der im *libhugin*–Prototyp implementierten „deep"–Strategie:

```
>>> from hugin.harvest.session import Session
>>> s = Session()
>>> q = s.create_query(title="drive", amount=7, strategy='deep')
>>> s.submit(q)
[<tmdbmovie <movie, picture> : Drive (2011)>,
 <tmdbmovie <movie, picture> : Drive (1998)>,
 <tmdbmovie <movie, picture> : Drive (2002)>,
 <ofdbmovie <movie> : Drive (2011)>,
 <ofdbmovie <movie> : Drive [Kurzfilm] (2011)>,
 <ofdbmovie <movie> : Drive (1997)>,
 <filmstartsmovie <movie> : Drive (2011)>]
```

Bei der *„flat"*–Strategie werden die Provider und Ergebnisse auf die gleiche Art wie bei der *„deep"*–Strategie gruppiert und sortiert. Anschließend werden aber jeweils die Ergebnisse mit der größten Übereinstimmung iterativ, angefangen beim Provider mit der höchsten Priorität, zurückgeliefert bis die gewünschte Anzahl erreicht ist.

Das folgende Beispiel zeigt das tatsächliche Ergebnis der im *libhugin*–Prototyp implementierten „flat"–Strategie:

```
>>> from hugin.harvest.session import Session
>>> s = Session()
>>> q = s.create_query(title="drive", amount=7, strategy='flat')
>>> s.submit(q)
[<tmdbmovie <movie, picture> : Drive (2011)>,
 <ofdbmovie <movie> : Drive (2011)>,
 <filmstartsmovie <movie> : Drive (2011)>,
 <omdbmovie <movie> : Drive (2011)>,
 <videobustermovie <movie> : Drive (2011)>,
 <tmdbmovie <movie, picture> : Drive (1998)>,
 <ofdbmovie <movie> : Drive [Kurzfilm] (2011)>]
```

Abbildung 5.7 visualisiert die Vorgehensweise der beiden Strategien.

5.6 Libhugin–harvest Plugins

Die bisher erläuterten Ansätze und Algorithmen werden direkt durch *libhugin* realisiert oder als Hilfsfunktionen bereitgestellt.

Des Weiteren wurden für den Prototypen Postprocessor–Plugins geschrieben, welche weitere Probleme der Metadatenbeschaffung angehen. Ob der Benutzer ein Plugin, beziehungsweise welche Plugins der Benutzer nutzen möchte, bleibt ihm überlassen.

Durch die einfach gestalteten Schnittstellen (vgl [1]) ist es möglich, *libhugin* um ein eigenes Plugin mit gewünschter Funktionalität zu erweitern.

Algorithmik der Postprocessor–Plugins

Das Postprocessor–Plugin „*Compose*" ist ein Plugin, welches es dem Benutzer erlaubt, verschiedene Metadatenquellen zusammenzuführen. Dies ist in der aktuellen Version auf zwei verschiedene Arten möglich.

1.) Das „automatische" Zusammenführen der Daten. Hierbei werden die gefundenen Suchergebnisse nach IMDb–ID gruppiert. Dies garantiert, dass die Metadaten nur zwischen gleichen Filmen ausgetauscht werden.

Findet der höchstpriorisierte Provider Metadaten zu einem Film, fehlt jedoch die Inhaltsbeschreibung, so wird diese durch den nächst niedriger priorisierten Provider ergänzt, der eine Inhaltsbeschreibung besitzt. Abbildung 5.8 zeigt die Funktionalität des *Compose*–Plugins. Zuerst wird eine

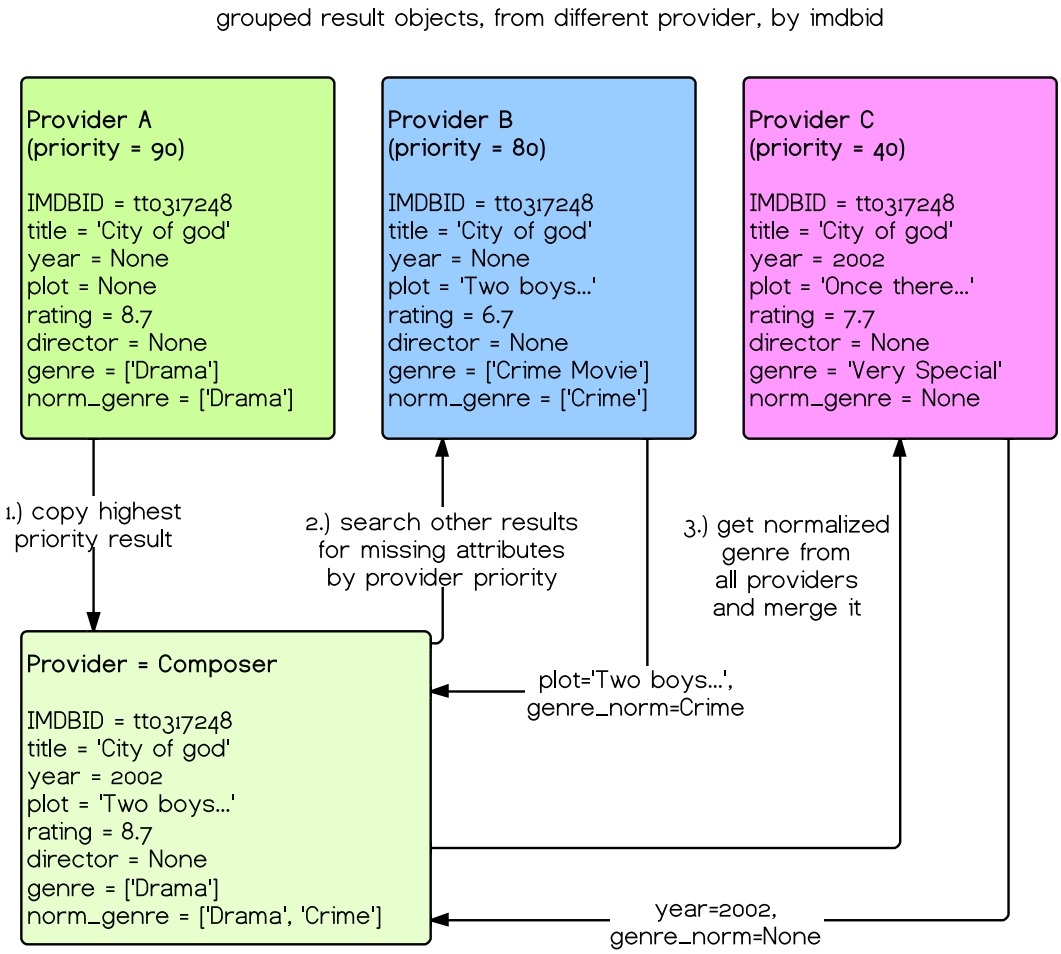

Abbildung 5.8.: Automatische Ergänzung fehlender Attribute mittels Compose-Plugin mit Genre Zusammenführung.

Ergebnisobjekt–Kopie vom Provider mit der höchsten Priorität erstellt, anschließend werden fehlende Attribute durch Attribute der anderen Ergebnisobjekte ergänzt, soweit diese vorhanden sind. Dabei erfolgt das Auffüllen der fehlenden Attribute *iterativ*, anfangend beim Provider mit der nächst niedrigeren Priorität. Dieser Ansatz funktioniert aktuell nur mit Onlinequellen, die eine IMDb–ID bereitstellen. Eine Erweiterung um Provider, die keine IMDb–ID bieten wäre möglich, indem hier zusätzliche Attribute wie beispielsweise der Regisseur herangezogen werden, um gleiche Filme zu gruppieren.

2.) Eine weitere Möglichkeit neben dem automatischen Zusammenführen von Attributen verschiedener Provider ist die Angabe einer benutzerdefinierten Profilmaske. Diese Profilmaske ist eine Hash–Tabelle mit den jeweiligen Attributen als Schlüssel und den gewünschten Providern als Wert. Folgende Python Notation gibt an, dass der Standardanbieter TMDb sein soll und die Inhaltsbeschreibung immer vom Provider OFDb befüllt werden soll.

Wenn dieser keine Inhaltsbeschreibung besitzt, soll das Ergebnis des OMDb–Provider genommen werden.

```
profile_mask = {
     'default':['tmdbmovie'],            # Grundkopie von TMDb
     'plot': ['ofdbmovie', 'omdbmovie']  # Plot von ofdb oder omdb
}
```

Nach dem Befüllen der fehlenden Attribute wird das Genre zusammengeführt. Dies passiert, indem die normalisierten Genres der verschiedenen Provider–Ergebnisse zu einer Liste aus Genres zusammengeführt werden.

Um die Postprocessor–Plugins vollständig zu benennen existiert noch ein *„Trim"*–Plugin. Dieses iteriert über alle Attribute eines Ergebnisobjektes und entfernt dabei mittels der Python `strip()`–Funktion die führenden und nachstehenden Leerzeichen.

Algorithmik der Converter–Plugins

Auf weitere Algorithmik, welche innerhalb der Converter–Plugins realisiert ist, wird aufgrund ihrer Einfachheit nicht weiter eingegangen. Hier werden jeweils nur Formatierungen der Ergebnisobjekte in ein bestimmtes Ausgabeformat wie beispielsweise XML [2] durchgeführt.

5.7 Libhugin–analyze Plugins

Der *libhugin–analyze* Teil der Bibliothek ist für das nachträgliche Bearbeiten von Metadaten gedacht. Insbesondere ist dieser Teil der Bibliothek konzipiert worden, um automatisiert große Filmsammlungen von mehreren hundert Filmen möglichst automatisiert mit wenig Aufwand pflegen zu können. Dabei werden die Daten mittels einer import/export–Funktion, die vom Benutzer bereitgestellt werden muss, in eine interne Datenbank importiert. Auf diesen Metadaten können dann Analysen sowie Modifikationen durchgeführt werden. Anschließend werden die modifizierten Daten mit Hilfe der vom Benutzer bereitgestellten import/export–Funktion wieder in das Produktivsystem exportiert. Für weitere Informationen und Anwendungsbeispiele siehe [1].

Algorithmik der Analyzer–Plugins

Die Analyzer–Plugins analysieren die Metadaten und schreiben die neu gewonnenen Informationen in eine dafür vorgesehene Liste. Die folgenden Analyzer–Plugins wurden im Prototypen implementiert:

[2] XML ist eine Auszeichnungssprache zur baumartig strukturierten Darstellung von Daten in Form von Textdateien.

Keywordextract–Plugin: Plattformen wie TMDb bieten neben den grundlegenden Metadaten wie Titel, Erscheinungsjahr et cetera auch Zusatzinformationen zu Filmen an. Ein Attribut, welches beim „Stöbern" oder der Auswahl eines Filmes hilfreich sein kann, sind Schlüsselwörter.

Alternativ zu Providern, die Schlüsselwörter für Filme anbieten, gibt es auch die Möglichkeit, Schlüsselwörter aus Texten automatisiert zu extrahieren. Hierzu gibt es verschiedene Algorithmen, jedoch werden hier zur Extraktion der Schlüsselwörter meistens sprachabhängige Korpora (Wort–Datenbanken) benötigt (vgl. [8]).

Ein weiterer Algorithmus, der ohne Korpus auskommt und dabei ähnlich gute Ergebnisse wie die korporabasierten Algorithmen liefert, ist der RAKE–Algorithmus (Rapid Automatic Keyword Extraction), vgl. [9], [10].

Hier wurde eine bereits existierende Implementierung in Kooperation mit dem Kommilitonen Christopher Pahl reimplementiert. Herr Pahl verwendet den Algorithmus zur Extraktion von Schlüsselwörtern aus Liedtexten, vgl. [11]. Der Algorithmus wurde um das automatische Laden einer *Stoppwortliste* und einen *Stemmer* erweitert.

Stoppwörter sind Wörter, die sehr häufig auftreten und somit keine Relevanz für die Erfassung des Dokumentinhalts besitzen. Libhugin verwendet hier die Stoppwortlisten verschiedener Sprachen von der Université de Neuchâtel [3].

Stemming ist ein Verfahren im Information Retrieval, bei dem die Wörter auf ihren gemeinsamen Wortstamm zurückgeführt werden.

Im Anschluß die Funktionsweise des RAKE–Algorithmus, analog zu [11]:

1. Aufteilung des Eingabetextes in Sätze anhand von Interpunktionsregeln.

2. Extrahieren von *Phrasen* aus den jeweiligen Sätzen. Eine *Phrase* ist eine Sequenz aus nicht Stoppwörtern.

3. Berechnung eines *Scores* für jedes Wort einer *Phrase* aus dem *Degree* und der *Frequency* eines Wortes.

   P entspricht der Menge aller Phrasen, $|p|$ ist die Anzahl der Wörter einer Phrase.

   $degree(word)$ gibt an, wie Häufig ein Wort pro Phrase vorkommt.

$$degree(word) = \sum_{p \in P} \begin{cases} |p|, & \text{falls } word \in p \\ 0, & \text{sonst} \end{cases}$$

[3] http://members.unine.ch/jacques.savoy/clef/index.htm

$frequency(word)$ ist der absolute Anteil der Phrasen in denen das jeweilige Wort vorkommt.

$$frequency(word) = \sum_{p \in P} \begin{cases} 1, & \text{falls } word \in p \\ 0, & \text{sonst} \end{cases}$$

4. Berechnung des *Scores* für jede Phrase. Dieser definiert sich durch die Summe aller Wörter–*Scores* innerhalb einer Phrase.

$$score(word) = \frac{degree(word)}{frequency(word)}$$

Für weitere Details zum RAKE–Algorithmus siehe [9].

Im Gegensatz zur Extraktion von Schlüsselwörtern aus Liedtexten werden bei der Extraktion aus der Film–Inhaltsbeschreibung die Sätzen nur anhand von Interpunktionsregeln getrennt, Zeilenumbrüche zählen hier nicht als Trennzeichen.

Folgende Inhaltsbeschreibung findet sich für den Film π (1998) auf TMDb:

> *Mathematikgenie Max Cohen steht kurz vor der Entschlüsselung eines numerischen Systems, das die Struktur von Zufall und Chaos aufdecken könnte. Mit diesem Code ließen sich nicht nur die Abläufe des Universums erklären, sondern auch Börsenbewegungen voraussagen. Bald sieht sich Max durch skrupellose Wall-Street-Haie verfolgt, aber auch eine religiöse Sekte und der Geheimdienst sind ihm auf den Fersen. Seine mentale Gesundheit leidet, er schlingert mehr und mehr in den Wahnsinn. Als es ihm gelingt, den 216-stelligen Code zu knacken, macht er eine Entdeckung, für die alle bereit sind, ihn zu töten...*

Abbildung 5.9 zeigt die relevanten (*Score* $>$ 1.0) Schlüsselwörter, die aus dem oben genannten Text, mittels RAKE–Algorithmus extrahiert wurden.

Im Vergleich zu den automatisch extrahierten Schlüsselwörtern sind auf der TMDb Plattform folgende Schlüsselwörter gepflegt:

> *hacker, mathematician, helix, headache, chaos theory, migraine, torah, börse, mathematics, insanity, genius*

FiletypeAnalyze–Plugin: Dieses Plugin dient dazu, Datei–Metadaten aus Filmdateien zu extrahieren. Da dies aufgrund der Vielzahl von Containern und Codecs ein nicht triviales Problem ist, implementiert der *libhugin–analyze* Prototyp diese Funktionalität mit Hilfe des Tools `hachoirmetadata`. Dieses Tool basiert auf der „Hachoir"–Bibliothek welche die Extraktion verschiedener Metadaten aus Multimedia–Dateien unterstützt. Das *FiletypeAnalyze*–Plugin führt das `Hachoir-metadata`–Kommandozeilen Tool aus, welches folgenden Output liefert:

Score	Schlüsselwörter
14.500	(‘mathematikgenie’, ‘max’, ‘cohen’, ‘steht’)
9.000	(‘mentale’, ‘gesundheit’, ‘leidet’)
4.000	(‘code’, ‘ließen’)
4.000	(‘börsenbewegungen’, ‘voraussagen’)
4.000	(‘chaos’, ‘aufdecken’)
4.000	(‘numerischen’, ‘systems’)
4.000	(‘haie’, ‘verfolgt’)
4.000	(‘universums’, ‘erklären’)
4.000	(‘stelligen’, ‘code’)
4.000	(‘religiöse’, ‘sekte’)
4.000	(‘skrupellose’, ‘wall’)
2.500	(‘max’)

Abbildung 5.9.: Extrahierte Schlüsselwörter aus der Inhaltsbeschreibung des Films Pi (1998).

```
$ hachoir-metadata --raw Sintel.2010.1080p.mkv
Common:
- duration: 0:14:48.032000
- creation_date: 2011-04-25 12:57:46
- producer: mkvmerge v4.0.0 ('The Stars were mine') built on Jun 17 2010 18:47:20
- producer: libebml v1.0.0 + libmatroska v1.0.0
- mime_type: video/x-matroska
- endian: Big endian
video[1]:
- width: 1920
- height: 818
- compression: V_MPEG4/ISO/AVC
audio[1]:
- title: AC3 5.1 @ 640 Kbps
- nb_channel: 6
- sample_rate: 48000.0
- compression: A_AC3
subtitle[1]:
- language: German
- compression: S_TEXT/UTF8
```

Diese Ausgabe wird vom Plugin betrachtet und die relevanten Informationen wie Auflösung, Laufzeit, et cetera extrahiert. Die Extraktion ist relativ einfach, da die `hachoir-metadata`–Ausgabe ein valides *Json*–Dokument ist, welches direkt in eine Python Hash–Tabelle umgewandelt werden kann. *Json* ist ein schlankes Dateiaustauschformat, ähnlich wie *XML*.

LangIdentify–Plugin: Dieses Plugin erkennt die Sprache des übergebenen Textes. Es ist für die Analyse der Sprache der Inhaltsbeschreibung gedacht. Mittels dem Plugin können große Filmsammlungen effizient analysiert werden und nicht vorhandene oder in einer unerwünschten Sprache gepflegte

Inhaltsbeschreibungen in wenigen Sekunden identifiziert werden. Das Plugin verwendet die Python–Bibliothek `guess_language-spirit`, welche die Sprache anhand von Sprachstatistiken erkennt. Die zusätzliche optionale Bibliothek `pyEnchant` kann von `guess_language-spirit` verwendet werden, um Texte mit weniger als 20 Zeichen zu erkennen. `Enchant` ist eine Bibliothek, welche auf verschiedene Sprachbibliotheken zugreifen kann.

Die folgende *IPython*–Sitzung zeigt die Funktionalität der Bibliothek:

```
>>> from guess_language import guess_language
>>> text = "Der Elfenkauz ist die einzige Art der Eulengattung der Elfenkäuze."
>>> guess_language(text)
'de'
```

Algorithmik der Modifier–Plugins

Die Modifier–Plugins modifizieren die Metadaten direkt. Hier wurde ein Plugin zum Bereinigen von Inhaltsangaben entwickelt, welches mittels regulärer Ausdrücke (vgl. [12]) unerwünschte, beispielsweise in Klammern stehende Inhalte, entfernt.

Die folgende *IPython*–Sitzung zeigt den Algorithmus im Einsatz:

```
>>> import re
>>> text  = "Die Elfenkäuzin (Micrathene Whitneyi) ist die einzige ihrer Gattung."
>>> re.sub('\s+\(.*?\)(\s*)', '\g<1>', text)
'Die Elfenkäuzin ist die einzige ihrer Gattung.'
```

Je nach Metadatenquelle finden sich hinter den jeweiligen Rollennamen, die Namen der Schauspieler in Klammern. Der Einsatz dieses Plugins soll eine einheitlichere Basis für weitere Untersuchungen der Inhaltsbeschreibung zwischen allen Metadatenquellen ermöglichen.

Algorithmik der Comparator–Plugins

Des Weiteren gibt es noch die experimentellen Comparator–Plugins, welche für den Vergleich von Metadaten untereinander gedacht sind. Dieser Teil ist im Prototypen noch nicht endgültig ausgebaut. Ziel ist es, hier über verschiedene Data–Mining–Algorithmen neue Erkenntnisse durch den Vergleich von Metadaten untereinander zu gewinnen, um beispielsweise Empfehlungen für ähnliche Filme aussprechen zu können.

Aktuell gibt es ein `KeywordCompare`–Plugin welches die Schlüsselwörter verschiedener Filme vergleicht, um eine Ähnlichkeit zu ermitteln. Der Ansatz, über Schlüsselwörter ähnliche Filme zu finden, hat bisher keine nennenswerten Erkenntnisse liefern können.

Das Comparator–Plugin `GenreCompare` versucht anhand vom Genre Ähnlichkeiten zwischen Filmen zu ermitteln. Die bisherigen Ergebnisse sind je nach verwendeter Metadatenquelle unterschiedlich

gut. Je feingranularer das Genre bei einem Anbieter gepflegt ist, umso *„ähnlicher"* ist die Grund–Thematik. Ein Film, der als Genre nur „Drama" gepflegt hat, kann zusätzlich in die Richtung Horror, Erotik, Thriller oder eine weitere nicht spezifizierte Richtung von der Handlung gehen.

Zusammenfassend kann gesagt werden, dass sich der Vergleich über das Genre zum aktuellen Zeitpunkt im Prototypen nur für die Eingrenzung der Filmauswahl auf ein bestimmtes Genre–Schema eignet.

6 | Analyse von Libhugin

Der *libhugin-harvest* Prototyp, der für die Beschaffung der Metadaten verwendet wird, hat aktuell die fünf Movie–Provider implementiert, siehe Abbildung 6.1.

	TMDb	OFDb	OMDb	Videobuster	Filmstarts
Zugriffsart	API	API	API	Scraping	Scraping
Sprache	multilingual	deutsch	englisch	deutsch	deutsch
Plattformtyp	Filmdatenbank	Filmdatenbank	Metadatenanbieter	Verleihdienst	Allg. Plattform

Abbildung 6.1.: Überblick implementierter Onlinequellen als Provider.

Des Weiteren wurden Personen–Provider für TMDb und OFDb implementiert.

6.1 Timeoutverhalten

Bereits während der Entwicklung ist bei der Erhebung der Daten aufgefallen, dass der OFDb–Provider kaum Metadaten findet. Nach kurzer Recherche war zu beobachten, dass hier der Zugriff über die API sehr oft einen Timeout mit der Fehlermeldung „`Fehler oder Timeout bei OFDB Anfrage`" liefert.

Eine genauere Analyse des Timeout–Verhaltens der Provider zeigt, dass die API vom OFDb–Provider sehr instabil ist. Hierzu wurden Metadaten für 100 Filme je Provider gezogen. Abbildung 6.2 zeigt wie oft es zu Fehlern pro Provider gekommen ist. Der Test wurde, um gegebenenfalls Server- oder Leitungsprobleme auszuschließen, an fünf verschiedenen Tagen durchgeführt. Für den Test wurde das Skript aus *Anhang F (Timeoutverhalten)* verwendet.

	TMDb	OFDb	OMDb	Videobuster	Filmstarts
Tag 1 (min/avg/max)	(0/0/0)	(0/31,87/60)	(0/0/0)	(0/0/0)	(0/0/0)
Tag 2 (min/avg/max)	(0/0/0)	(0/0.87/6)	(0/0/0)	(0/0/0)	(0/0/0)
Tag 3 (min/avg/max)	(0/0/0)	(0/1.23/13)	(0/0/0)	(0/0/0)	(0/0/0)
Tag 4 (min/avg/max)	(0/0/0)	(0/4.61/55)	(0/0/0)	(0/0/0)	(0/0/0)
Tag 5 (min/avg/max)	(0/0/0)	(0/3.56/55)	(0/0/0)	(0/0/0)	(0/0/0)

Abbildung 6.2.: Anzahl der „retries" beim Herunterladen von Metadaten für jeweils 100 Filme.

Der OFDb–Provider verteilt die Anfragen über einen Lastverteiler, siehe [Link–11]. Während der Entwicklung hat eine Stichprobe mit 10 Filmen gezeigt, dass Anfragen über den Lastverteiler zu

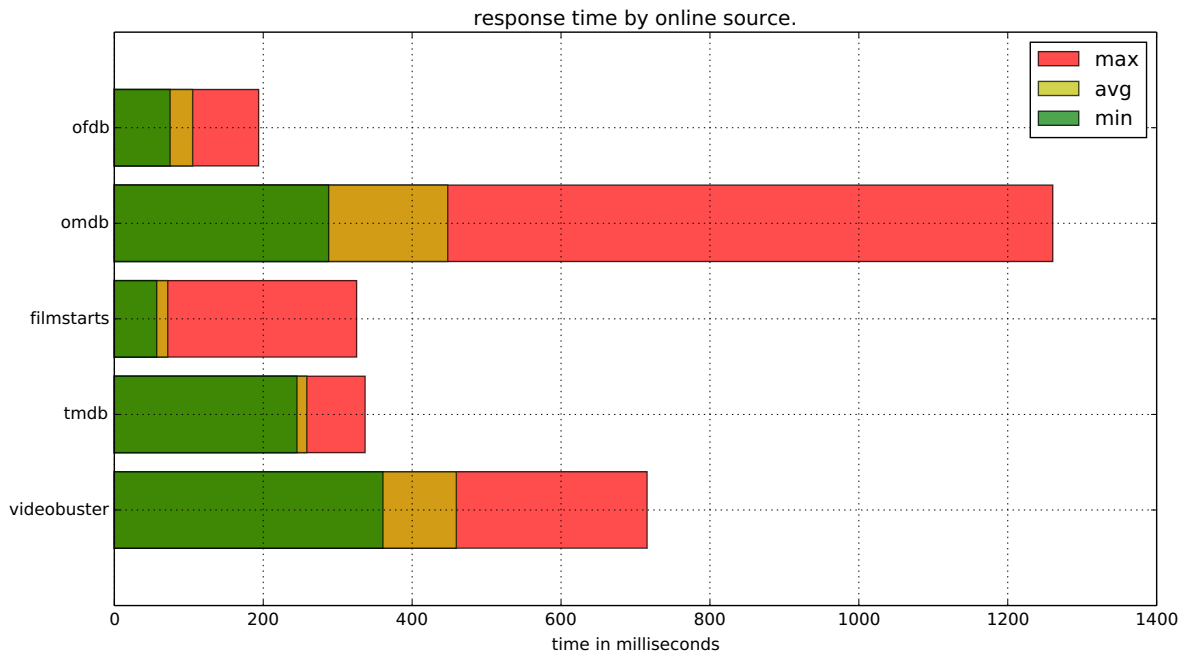

Abbildung 6.3.: Antwortzeiten der vom libhugin Prototypen verwendeten Onlineplattformen im Überblick. Minimum (grün), Durchschnitt (gelb), Maximum (rot). Das jeweilige Balkenende repräsentiert den exakten Wert.

unvollständigen Ergebnissen führten. Hier wurden die Filme ohne Inhaltsbeschreibung zurückgeliefert.

Ein Testen der einzelnen Server ergab, dass `http://ofdbgw.geeksphere.de` als einziger Mirror die erwarteten Ergebnisse lieferte. Dieser wurde somit im Prototypen direkt als einziger Server aktiviert. Weitere Analysen der Metadaten sollen Aufschluß darüber geben, ob das Problem weiterhin auftritt.

6.2 Antwortzeiten der Onlinequellen

Abbildung 6.3 zeigt die Antwortzeiten der jeweiligen Plattformen/Metadatenanbieter, die *libhugin* als Provider implementiert hat. Hierbei wurde jeweils die „Suchseite" des jeweiligen Anbieters angefordert. Die Zeit wurde mit dem Skript im *Anhang G (Antwortzeiten Http Metadatenquellen)* gemessen.

Der Zugriff in Abbildung 6.3 zeigt hier den direkten Zugriff über die HTTP–Bibliothek. Bei *libhugin–harvest* besteht die Standardsuche (über Titel) nach Metadaten in der Regel aus mehreren Zugriffen (siehe Abbildung 6.4). Zusätzlich kommt hier noch der Aufwand für das Extrahieren der Metadaten aus den jeweiligen HTTP–Response Objekten hinzu.

Bei der Suche nach Metadaten für einen Film haben die Provider jeweils einen Zugriff für die Suchanfrage und einen weiteren Zugriff für den jeweiligen Film.

	TMDb	OFDb	OMDb	Videobuster	Filmstarts
Anzahl der Zugriffe	2	2	2	2	3

Abbildung 6.4.: Anzahl der Zugriffe bei der Standardsuche.

Der Filmstarts Provider benötigt bei Zugriff auf den jeweiligen Film zwei Suchanfragen (siehe Abbildung 6.4), da auf dieser Plattform die Schauspieler–Informationen zum Film auf einer separaten Seite zu finden sind.

Folgende Auflistung zeigt die angesprochenen Seiten des Filmstarts–Providers:

Suchanfrage nach Metadaten für Film „*The Matrix*":

1. `http://www.filmstarts.de/suche/?q=the+matrix`

Zugriff auf Seiten mit Metadaten zum Film „*The Matrix*":

1. `http://www.filmstarts.de/kritiken/35616-Matrix.html`
2. `http://www.filmstarts.de/kritiken/35616-Matrix/castcrew.html`

6.3 Antwortzeiten der Libhugin–Provider

Abbildung 6.5 zeigt die Geschwindigkeit beim Zugriff auf Metadaten über die *libhugin–harvest*–Bibliothek. Hier wurde *libhugin–harvest* so konfiguriert, dass pro Provider einzeln jeweils 10 Filme heruntergeladen werden. Das Ergebnis ist jeweils der Durchschnitt aus 10 Durchläufen. Das Skript in *Anhang H (Antwortzeiten Libhugin Metadatenquellen)* wurde für diesen Benchmark verwendet.

Auffällig ist hier die fast doppelt so lange Zeit bei den Providern ohne API.

Eine zweite Auswertung mit den gleichen Daten und aktivierten Festplatten–Cache (Metadaten werden von der Festplatte geladen, es findet kein Webzugriff statt) zeigt, dass die Provider mit API im Gegensatz zu den Providern ohne API die Metadaten in sehr kurzer Zeit verarbeiten.

Die auffällige Antwortzeit mit aktivierten Festplatten–Cache (Abbildung 6.6) deutet darauf hin, dass das Extrahieren der Metadaten mittels der `Beautiful-Soup`–Bibliothek sehr aufwendig ist. Das Aktivieren eines anderen internen Parsers hat das Ergebnis verschlechtert. Der *lxml*–Parser, welcher auch in Abbildung 6.6 verwendet wird, ist hier schneller als mögliche Alternativen (siehe [Link–12]).

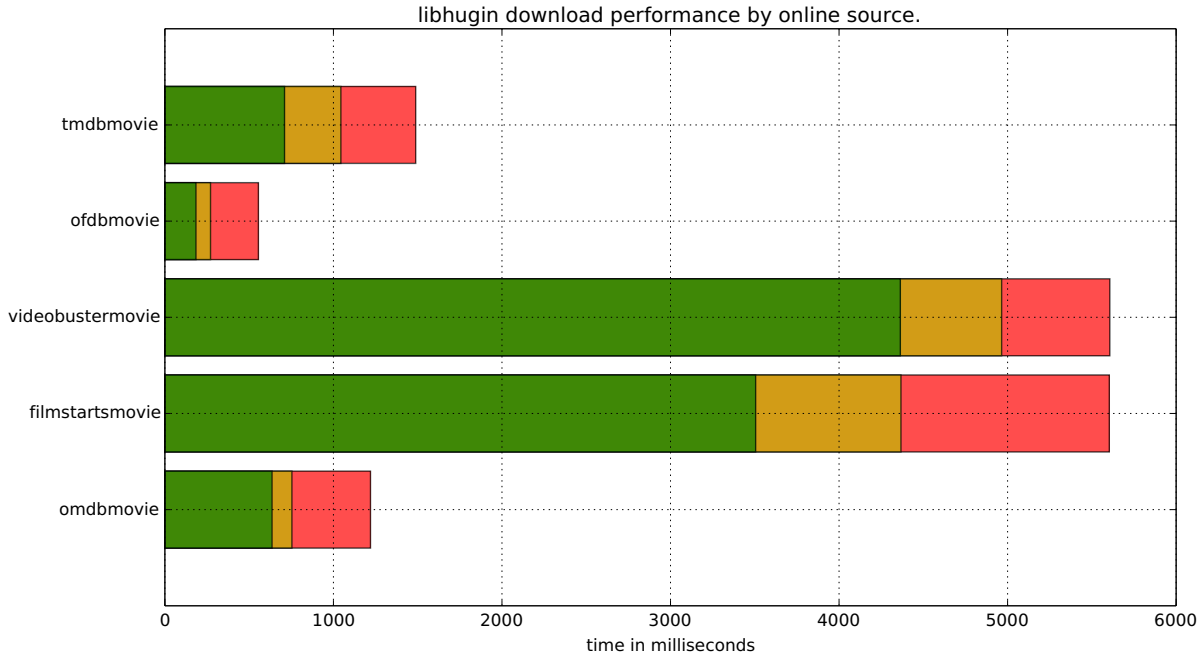

Abbildung 6.5.: Downloadgeschwindigkeit der Metadaten für einen Film pro Provider mit libhugin-harvest. Durchschnitt aus 10 verschiedenen Filmen. Minimum (grün), Durchschnitt (gelb), Maximum (rot). Das jeweilige Balkenende repräsentiert den jeweiligen Wert.

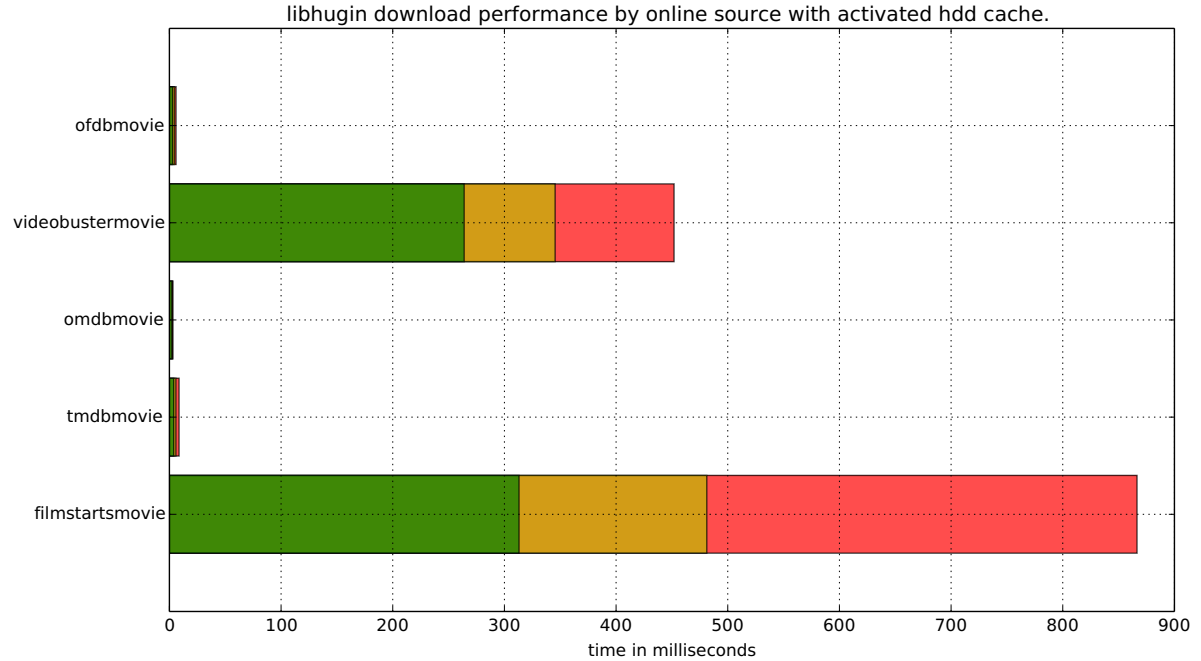

Abbildung 6.6.: Downloadgeschwindigkeit der Metadaten für einen Film pro Provider mit libhugin-harvest mit aktiviertem Cache. Durchschnitt aus 10 verschiedenen Filmen. Minimum (grün), Durchschnitt (gelb), Maximum (rot). Das jeweilige Balkenende repräsentiert den jeweiligen Wert.

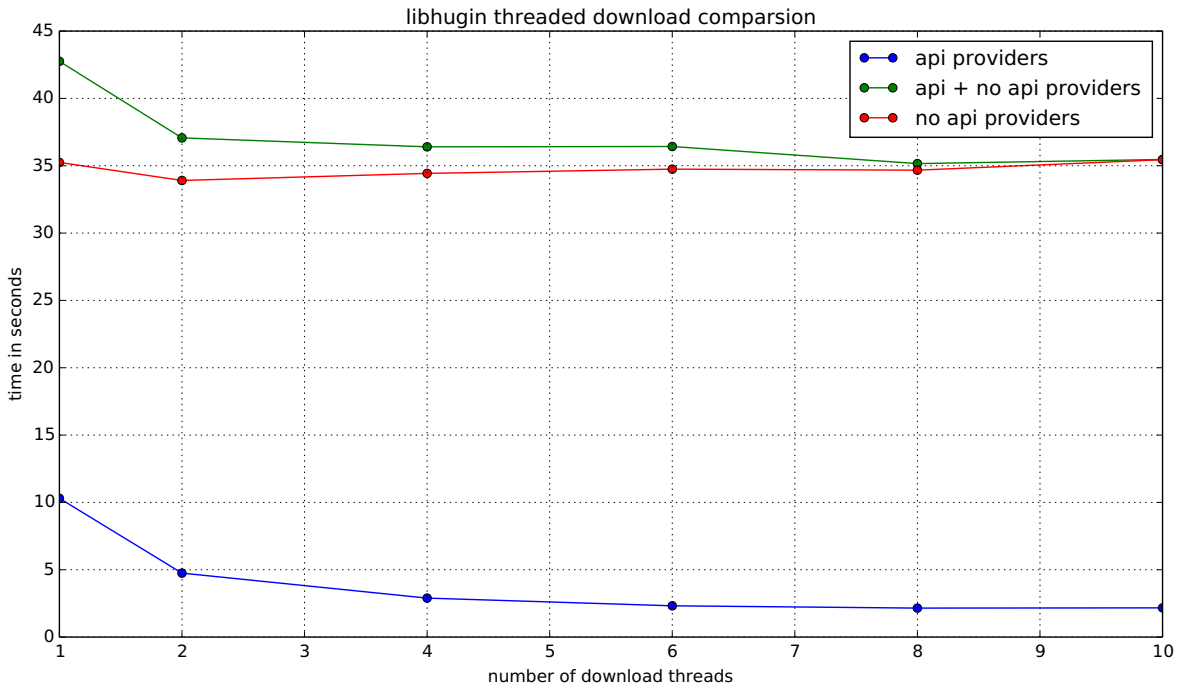

Abbildung 6.7.: Suche nach dem Film „Sin" mit einer unterschiedlichen Anzahl von Download-Threads. Die Ergebnisanzahl wurde auf 10 beschränkt. Das heisst, jeder Provider zieht maximal 10 Filme.

6.4 Skalierung der Downloadgeschwindigkeit

Abbildung 6.7 zeigt das Herunterladen von Metadaten mit einer unterschiedlichen Anzahl von parallelen Downloads. Hier wurden jeweils separat die API und non–API Provider ausgewertet, um genauere Aussagen über die Effizienz beim parallelen Herunterladen machen zu können.

Bei den API–Provider ist eine signifikante zeitliche Verbesserung mit steigender Download–Thread Anzahl erkennbar. Hier ist die Zeit von ca. 9 Sekunden auf 2 Sekunden gefallen (siehe Abbildung, 6.8).

Die non–API Provider bremsen die Performance aufgrund des aufwendigen Extrahierens mittels `Beautiful-Soup`–Bibliothek stark aus. Hier bewegt sich die Zeit zwischen 35 – 42 Sekunden für die Beschaffung von 10 Ergebnissen.

Die theoretischen Annahmen über die Skalierung der Downloadgeschwindigkeit aus Kapitel 4 werden mit der Einschränkung auf die Limitierung der non–API Provider bestätigt.

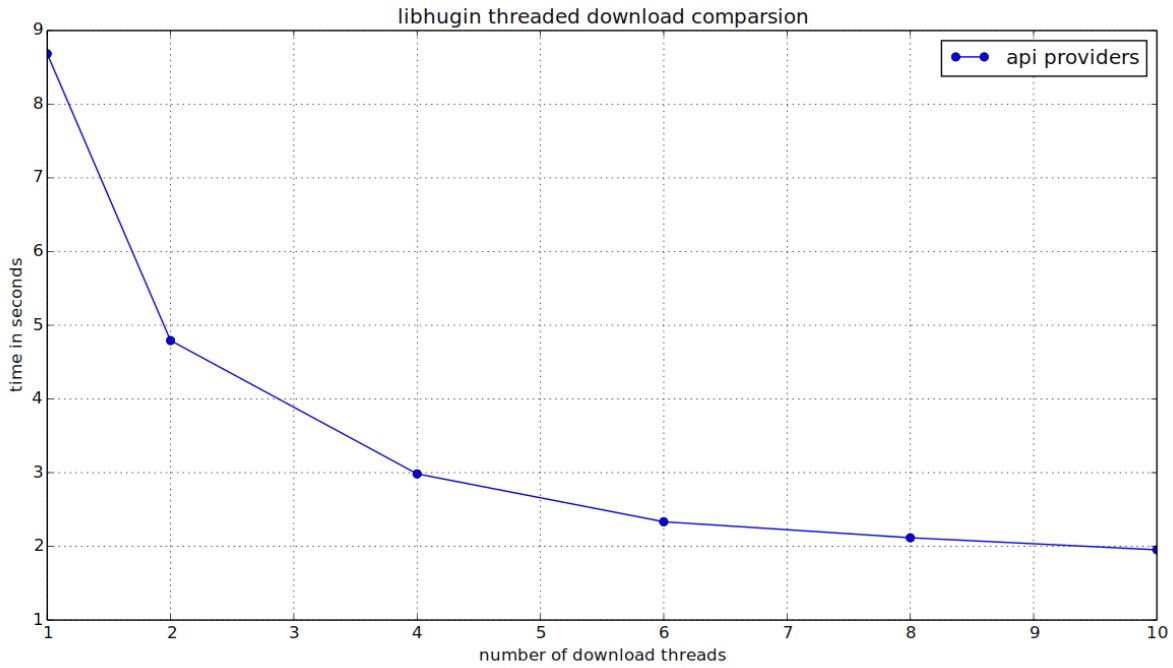

Abbildung 6.8.: Suche nach dem Film „Sin" mit einer unterschiedlichen Anzahl von Download-Threads. Die Ergebnisanzahl wurde auf 10 beschränkt. Das heisst, jeder Provider zieht maximal 10 Filme.

Die Auswertung der Skalierung der Downloadgeschwindigkeit wurde mit dem Skript *Anhang I (Libhugin Threaded Downloadgeschwindigkeit)* durchgeführt.

7 | Analyse der Metadaten

Die im Prototypen implementierten Metadatenquellen weisen unterschiedliche Eigenschaften auf. Allgemein und auch für die Entwicklung des Prototypen wurden bestimmte Annahmen getroffen:

- Starke Unterschiede in der Genre–Verteilung zwischen den Quellen.
- Starke Unterschiede im Genre–Detailgrad zwischen den Quellen.
- Erscheinungsjahr–Differenzen bei gleichen Filmen zwischen den verschiedenen Quellen.
- Unvollständigkeit der Metadaten vieler Filme.
- Bewertungsverteilung der verschiedenen Quellen variiert stark.

Diese Annahmen sollen folgend anhand einer Stichprobe untersucht werden.

7.1 Testdatenbeschaffung

Für die Analyse der Metadaten wurde eine Metadaten–Stichprobe von 2500 Filmen mit Hilfe der *libhugin-harvest*–Bibliothek beschafft. Die Zusammenstellung besteht aus möglichst zufällig gewählten Filmen verschiedener Kategorien. Es ist grundsätzlich schwierig, eine „optimale" Metadaten–Stichprobe auszusuchen, da die Plattformen unterschiedliche Ziele verfolgen.

Abbildung 7.1 zeigt die Verteilung der Filme anhand vom Erscheinungsjahr.

Für die Beschaffung der Metadaten wurden die IMDb–IDs von 2500 Filmen in einer Datei gesammelt. Anschließend wurden über ein IMDb–Lookup–Skript (siehe *Anhang J (IMDB Title Lookup)*) alle deutschsprachigen Titel und Erscheinungsjahre anhand der IMDb–ID bezogen. Mit diesen Informationen wurden 2500 Ordner mit der Struktur `[Filmtitel;Erscheinungsjahr;Imdbid]` angelegt, hierzu wurde das gleiche Skript verwendet.

Anschließend wurden die Metadaten mit Hilfe von *libhugin–harvest* über die fünf genannten Provider bezogen. Hierbei wurden die Metadaten bei den Providern mit IMDb–ID Unterstützung über diese bezogen. Provider, die keine IMDb–ID Unterstützung besitzen, wurden über den über IMDb „normalisierten" deutschen Titel mit Erscheinungsjahr bezogen. Die Metadaten wurden ebenso mit dem Skript *Anhang J (IMDB Title Lookup)* bezogen. Ein komprimiertes Archiv mit den Testdaten findet sich unter [Link–13].

Erscheinungsjahr	Anzahl	Erscheinungsjahr	Anzahl	Erscheinungsjahr	Anzahl
2013	53	2001	76	1989	15
2012	224	2000	57	1988	13
2011	253	1999	50	1987	10
2010	244	1998	55	1986	13
2009	245	1997	48	1985	12
2008	226	1996	27	1984	15
2007	194	1995	40	1983	7
2006	135	1994	23	1982	10
2005	118	1993	18	1981	4
2004	109	1992	19	1980	9
2003	77	1991	12	1979	4
2002	74	1990	11		

Abbildung 7.1.: Testdaten nach Erscheinungsjahr.

Die API basierten Provider haben jeweils 2500 Filme gefunden. Bei den Provider ohne API wurden ca. 2-3 % nicht gefunden, siehe dazu Abbildung 7.2.

	tmdb	ofdb	omdb	videobuster	filmstarts
gefundene Filme	2500	2500	2500	2444	2427
Suche über IMDBID	✓	✓	✓	×	×
Onlinezugriff über API	✓	✓	✓	×	×

Abbildung 7.2.: Überblick Metadatensuche für 2500 Filme.

Eine Stichprobe von jeweils fünf nicht gefundenen Filmen von Videobuster und Filmstarts wurde genauer betrachtet:

Filmstarts:

- „Secretary (2002)", wird ohne Titelzusatz gefunden.
- „Reservoir Dogs (1992)", wird ohne Titelzusatz gefunden.
- „Peter & der Wolf (2006)", auf Plattform nicht vorhanden.
- „One Dark Night (1982)", auf Plattform nicht vorhanden.
- „O Brother, Where Art Thou? (2000)", wird ohne Titelzusatz gefunden.

Videobuster:

- „Mimic (1997)", wird ohne Titelzusatz gefunden.
- „Miez und Mops (1986)", auf Plattform nicht vorhanden.

- „Like Someone in Love (2012)", auf Plattform nicht vorhanden.
- „The Last House on the Left (2009)", wird wegen Altersverifikation nicht gefunden.
- „Infernal Affairs (2002)", wird ohne Titelzusatz gefunden.

Anmerkung zum Titelzusatz: Die über IMDb „normalisierten" Titel haben oft einen Titelzusatz. Beispielsweise der Film „Secretary (2002)" wurde über IMDb auf „Secretary – Womit kann ich dienen? (2002)" normalisiert.

Der Stichprobe nach zu urteilen gibt es hier bei Videobuster und Filmstarts Probleme. Bei der Suche nach dem Filmtitel ohne Titelzusatz werden die Titel gefunden, falls vorhanden.

Die Stichprobe der 10 Filme zeigt, dass die nicht gefundenen Filme durchaus auf der jeweiligen Plattform gepflegt sein können.

7.2 Analyse der Genreinformationen

Das Genre unterscheidet sich oft bei den gepflegten Plattformen. Das liegt daran, dass das Genre an sich nicht standardisiert ist und die Onlineplattformen teils divergente Genre–Bezeichnungen haben. Die folgenden Auswertungen sollen den Umstand anhand der gewählten Stichprobe, sowie alle bisher für die Entwicklung getroffenen Annahmen, bestätigen.

Die Daten in Abbildung 7.3 wurden mit dem Skript im *Anhang K (Genre Analyse)* erhoben und zeigen die Genreverteilung der fünf Provider für die Metadaten der 2500 Filme. Bei Filmstarts beziehen sich die Genreinformationen lediglich nur auf 2427 Filme, bei Videobuster nur auf 2444 Filme.

OFDb/2500	OMDb/2500	TMDb/2500	Videobuster/2444	Filmstarts/2427
Abenteuer: 180	Action: 650	Abenteuer: 362	18+ Spielf.: 332	Abenteuer: 202
Action: 609	Adult: 2	Action: 753	Abenteuer: 113	Action: 529
Biographie: 60	Adventure: 331	Animation: 124	Action: 395	Animation: 112
Dokumentation: 33	Animation: 125	Dokumentarf.: 36	Animation: 98	Biografie: 50
Drama: 1086	Biography: 104	Drama: 1200	Anime: 24	Dokumentation: 43
Eastern: 4	Comedy: 722	Eastern: 2	Bollywood: 2	Drama: 801
Erotik: 26	Crime: 575	Erotik: 6	Deutscher F.: 127	Erotik: 22
Essayfilm: 1	Documentary: 33	Familie: 130	Dokumentation: 38	Experimentalf.: 1
Experimentalf.: 1	Drama: 1239	Fantasy: 182	Drama: 616	Familie: 50
Fantasy: 193	Family: 76	Film Noir: 2	Fantasy: 180	Fantasy: 229
Grusel: 5	Fantasy: 169	Foreign: 152	Horror: 304	Gericht: 8
Heimatfilm: 1	History: 48	Historie: 52	Kids: 47	Historie: 46
Historienf.: 19	Horror: 349	Holiday: 1	Komödie: 491	Horror: 313
Horror: 352	Music: 31	Horror: 387	Kriegsfilm: 47	Komödie: 578
Kampfsport: 16	Musical: 12	Indie: 149	Krimi: 275	Kriegsfilm: 37
Katastrophen: 8	Mystery: 264	Katastrophenf.: 4	Lovestory: 142	Krimi: 209
Familienfilm: 110	Romance: 317	Komödie: 718	Musik: 31	Martial Arts: 16
Komödie: 727	Sci-Fi: 258	Kriegsfilm: 57	Ratgeber: 1	Monumentalf.: 3
Krieg: 56	Short: 10	Krimi: 452	Science-Fiction: 223	Musical: 7
Krimi: 193	Sport: 38	Lovestory: 341	Serie: 17	Musik: 28
Liebe/Romantik: 257	Thriller: 650	Musical: 23	Softerotik: 1	Romanze: 216
Musikfilm: 30	War: 37	Musik: 23	TV-Film: 10	Sci-Fi: 235
Mystery: 79	Western: 6	Mystery: 239	Thriller: 599	Spionage: 29
Science-Fiction: 271		Neo-noir: 3	Western: 15	Sport: 1
Sex: 5		Road Movie: 3		Thriller: 671
Splatter: 34		Science Fiction: 337		Tragikomödie: 127
Sportfilm: 31		Short: 6		Unbekannt: 25
Thriller: 803		Sport: 15		Western: 11
Tierfilm: 8		Sport Film: 12		Kein Genre: 1
Western: 10		Suspense: 53		
		Thriller: 1000		
		Western: 10		
		Kein Genre: 25		

Abbildung 7.3.: Überblick Unterschiede in der Genreverteilung bei ca. 2500 Filmen.

Beim TMDb und Videobuster Provider war das Genre Komödie auf jeweils drei Genre aufgrund eines fehlerhaften Encoding verteilt. Dieser Umstand wurde per Hand korrigiert. Des Weiteren wurden vereinzelt Genres abgekürzt, um die Tabelle darstellen zu können (f./F. $\hat{=}$ Film).

Aus Abbildung 7.3 ist nur schwer ersichtlich wie sich die Genreinformationen im Schnitt pro Film verteilen, beziehungsweise wie detailliert die Filme im Schnitt gepflegt sind. Abbildung 7.4 zeigt wie detailliert die Genreverteilung im Schnitt pro Film ist.

Genres pro Film	OFDb	OMDb	TMDb	Videobuster	Filmstarts
0	0	0	25	0	1
1	701	372	398	976	913
2	1029	713	666	1259	926
3	639	1412	783	202	522
4	123	3	435	7	57
5	8	0	153	0	8
6	0	0	30	0	0
7	0	0	10	0	0
Durchschnittlich	**2,08**	**2,42**	**2,73**	**1,69**	**1,89**

Abbildung 7.4.: Anzahl der vergebenen Genres pro Film.

Die Auswertung bestätigt die bisherigen Annahmen. Die Genreinformationen sind hier sehr divergent (siehe Abbildung 7.3) gepflegt und unterscheiden sich auch im Detailgrad (siehe Abbildung 7.4).

7.3 Analyse der Erscheinungsjahrdifferenz

Bei der Entwicklung wurde aufgrund der persönlichen Erfahrung des Autors die Algorithmik beim Zeichenkettenvergleich so angepasst, damit das Erscheinungsjahr „einzeln" betrachtet wird. Hier wurde bisher davon ausgegangen, dass es zwischen den Plattformen beim Erscheinungsjahr immer wieder zu Differenzen von ein bis zwei Jahren kommen kann.

Die erhobenen Metadaten wurden dahingehend mit dem Skript im *Anhang L (Differenz Erscheinungsjahr)* analysiert. Hier werden für die Betrachtung die API–Provider und die non–API–Provider hergenommen. Bei den API–Providern wird die Gleichheit des Films anhand der IMDb–ID definiert. Bei den non–API–Provider–Daten, die keine IMDb–ID besitzen, wird eine Titelübereinstimmung von 90% gefordert. Filme, die diese Eigenschaft erfüllen, fließen in die Erscheinungsjahrdifferenz–Auswertung ein (siehe Abbildung 7.5). Als Bezugsreferenz wurde hier der TMDb Provider genommen.

Die Videobuster und Filmstarts Ergebnisse wurden zusätzlich manuell auf die Übereinstimmung des Regisseurs überprüft. Hier wurde eine Übereinstimmung des Namens von 95% gefordert. Dieser

Jahresdifferenz zu TMDb:	OFDb	OMDb	Filmstarts	Videobuster
0 Jahre	2378	2403	1844	1792
1 Jahre	109	87	198	118
2 Jahre	8	5	13	8
3 Jahre	2	2	3	3
> 3 Jahre	0	0	42	36

Abbildung 7.5.: Überblick der unterschiedlich gepflegten Erscheinungsjahre gleicher Filme.

stimmt in insgesamt 317 von 343 (1 - 3 Jahre) Fällen überein. In den restlichen 26 Fällen war in 13 Fällen ein Vergleich nicht möglich, in weiteren 13 war der Film unterschiedlich.

Die restlichen insgesamt 78 Filme, die bei der Jahresdifferenz > 3 gelistet sind, wurden manuell auf Regisseur Übereinstimmung untersucht. Hier gab es nur eine einzige Übereinstimmung, die restlichen 77 Filme waren „Remakes", Filme mit zufälligerweise gleichem Titel oder Filme ohne gelisteten Regisseur.

7.4 Unvollständigkeit der Metadaten

Abbildung 7.6 zeigt die Anzahl der nicht gepflegten Attribute je Provider. Die Menge bezieht sich hier auf die pro Provider jeweils gefundene Anzahl der Metadaten (siehe Abbildung 7.2). Die mit $\times$ markierten Felder deuten darauf hin, dass das Attribut vom Provider nicht ausgefüllt wird.

Auffällig in Abbildung 7.6 ist, dass der OFDb–Provider das Attribut „plot" 2353 mal nicht gefunden hat. Die manuelle Überprüfung dieses Wertes bestätigt, dass es hier bei dem verwendeten API–Mirror, wie bereits erwähnt in Kapitel 6.1 Timeoutverhalten, entgegen der vorherigen Annahme, weiterhin zu Problemen kommt. Die Daten wurden mit dem Skript *Anhang M (Unvollständigkeit Metadaten)* analysiert.

Die Abbildung 7.6 zeigt, dass je nach Onlinequelle die Vollständigkeit der Metadaten nicht gewährleistet werden kann. Es zeigt ebenso, dass Plattformen wie Videobuster das Attribut „Poster/Cover" vollständig gepflegt haben. Bei diesem Anbieter handelt es sich um eine Videoverleihplattform, welche anscheinend darauf Wert legt, dass jeder ausleihbare Film auch ein digitales Cover besitzt.

Attribute	OFDb	OMDb	TMDb	Videobuster	Filmstarts
title	0	0	0	0	0
original_title	0	0	0	0	×
plot	2353	57	81	5	151
runtime	×	30	×	×	×
imdbid	0	0	0	×	×
vote_count	5	0	101	×	×
rating	0	0	482	×	×
alternative_titles	×	×	315	×	×
directors	0	4	19	8	109
writers	2404	12	1818	×	×
year	0	1	2	0	5
poster	0	82	707	0	1
fanart	×	×	2465	×	×
countries	0	×	104	0	×
genre	0	0	25	0	1
studios	×	×	434	0	×
actors	132	6	23	137	442
keywords	×	×	444	129	×
tagline	×	×	1833	1138	×

Abbildung 7.6.: Überblick fehlende Metadaten

7.5 Ratingverteilung der Stichprobe

Folgend findet sich eine Rating–Auswertung zu den drei API–basierten Providern. Die non–API–basierten Provider befüllen in der aktuellen Version das Attribut Rating nicht.

Die Analyse soll darüber Auskunft geben, ob es bei den Plattformen in der Bewertung signifikante Unterschiede gibt. Bei allen drei Anbietern bewegt sich das Rating auf einer Skala von 0 – 10.

Abbildung 7.7 zeigt, dass das Rating der Stichprobe bei allen drei Providern sich im Schnitt bei ca 6,5 von 10 bewegt.

	OMDb	TMDb	OFDb
Minimales Rating in der Stichprobe	1.9	0.2	0
Durchschnittliches Rating der Stichprobe	6.57	6.36	6.46
Maximales Rating der Stichprobe	10.0	10.0	9.0

Abbildung 7.7.: Ratingverteilung der Stichprobe.

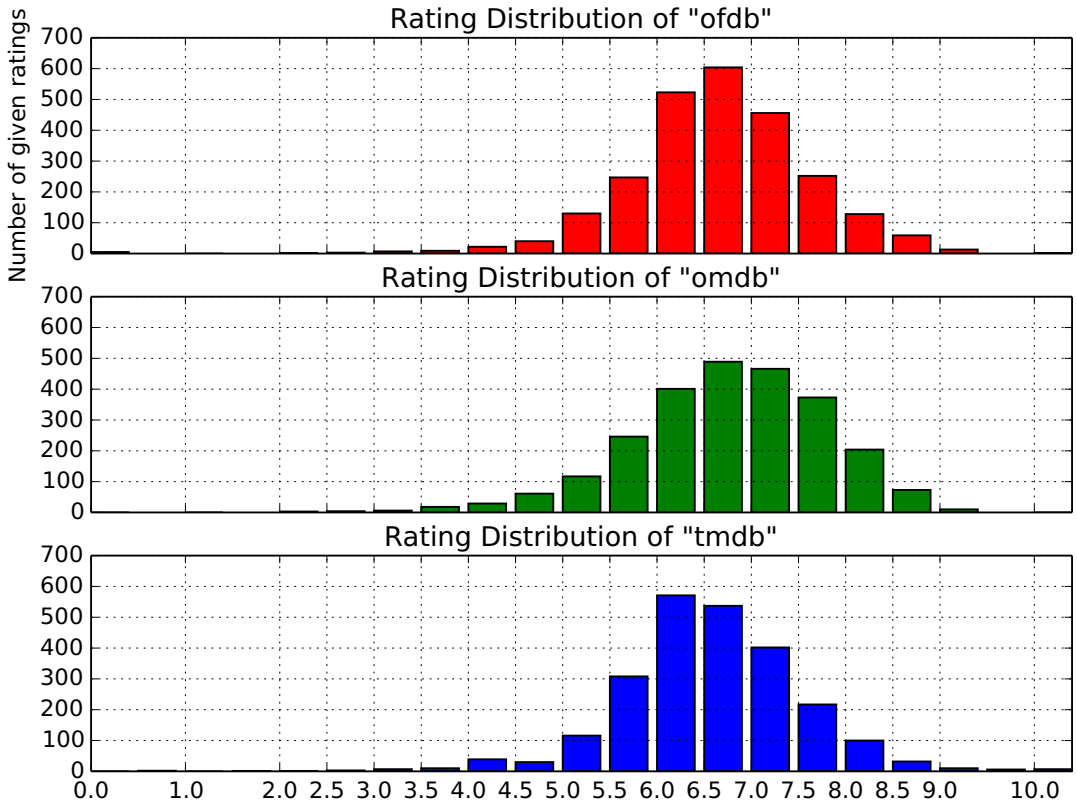

Abbildung 7.8.: Verteilung von der Filmbewertung der Stichprobe von 2500 Filmen der drei Anbieter TMDb, OMDb und OFDb.

Die Abbildung 7.8 zeigt, dass die Verteilung der Filmbewertung der drei API–Provider bei allen drei Onlinequellen sehr ähnlich ausfällt.

Die vorliegenden Daten wurden mit dem Skript in *Anhang N (Ratingverteilung)* analysiert.

8 | Trivia

8.1 Testumgebung

Die Bibliothek wurde in der Python–Version 3.4 getestet. Die Skripte im Anhang wurden für die jeweiligen Auswertungen verwendet. Für das Einlesen der Metadaten verwenden manche Skripte die Funktion analyze_folder. Diese Funktion wurde in die `utils.py`–Datei ausgelagert (siehe *Anhang O (Utilities)*).

Bei zeitabhängigen Messungen wurde darauf geachtet, dass immer der Durchschnitt aus mehreren Durchläufen genommen wurde, um statistische Ausreißer zu unterdrücken.

Als Testumgebung wurde das folgende System verwendet:

- OS: Arch Linux, 3.14.6-1-ARCH x86_64 (64 bit)
- CPU: Intel Core 2 Quad Q6600 @ 2.40GHz
- RAM: 4 GB DDR2 RAM
- HDD: Hitachi 120GB, 5400 upm

Als Internetanbindung wurde eine VDSL 50 Mbit Leitung der Telekom verwendet. Diese hat laut Internet–Messverfahren eine gemittelte Geschwindigkeit von ungefähr 48 Mbit/s (downstream) und 8 Mbit/s (upstream).

8.2 Statistiken und Plots

Für das Analysieren der Metadaten wurden eigene Skripte geschrieben. Diese sind an jeweiliger Stelle genannt und befinden sich im Anhang. Für das Erstellen der Grafiken/Plots wurde die Python Matplotlib–Bibliothek verwendet (siehe [Link–14]).

9 | Zusammenfassung

9.1 Verwendete Algorithmik

Die Evaluation der Bibliothek und der Metadaten konnte die bisher getroffenen Annahmen bestätigen. Das Downloadverhalten skaliert über mehrere Threads gut. Die Algorithmik für den Zeichenkettenvergleich wurde angepasst, um auch Filmtitel mit nachziehenden Artikeln zu finden. Hier skaliert der Algorithmus basierend auf der Damerau–Levenshtein–Distanz weiterhin gut (siehe Kabitel 5.1 Abbildung 5.4). Die zusätzliche Gewichtung vom Erscheinungsjahr erwies sich mit der kleinen Testdatenmenge als wirkungsvoll (siehe Kapitel 5.1, Abbildung 5.5).

Der verwendete Ansatz beim „Lookup"–Mode funktioniert ebenso wkirkungsvoll. Dieser Ansatz wurde zum Normalisieren der Filmtitel für die Metadaten–Stichprobe verwendet.

Weitere Ansätze wie die Implementierung der Unschärfesuche können nur schwer beurteilt werden. Hier gibt es je falsch geschriebenen Titel Toleranzen. Im Grunde kommt es hier auf die Genauigkeit der Suchmaschine an. Kleine Stichproben in der Projektarbeit zeigten eine hohe Erfolgsquote (siehe [1], Kapitel 7.4 Demoanwendungen).

Die verwendeten Algorithmen bei den Plugins greifen aktuell zum Teil auf externe Tools zurück, wie beispielsweise das *FileTypeAnalyze* Plugin. Hier wären in naher Zukunft Ansätze wünschenswert, die sich mehr in die Bibliothek integrieren.

9.2 Untersuchungen der Metadaten

Die Annahmen über die Metadaten wurden mit einer Stichprobe von 2500 Filmen fast vollständig bestätigt. Das Genrespektrum sowie Gewichtung bei den Onlinequellen ist hier sehr unterschiedlich (siehe Kapitel 7.2, Abbildung 7.3). Des Weiteren variiert der Detailgrad des Genres pro Film, je nach Datenquellen mehr oder weniger stark. Durchschnittlich kommt es hier zu Abweichungen von mehr als einem Genre (siehe Kapitel 7.2, Abbildung 7.4).

Die Annahme und persönliche Erfahrung des Autors, dass es hier Differenzen beim Erscheinungsjahr gibt, wurden bestätigt (siehe Kapitel 7.3, Abbildung 7.5).

Die Annahme, dass die Metadaten lückenhaft gepflegt sind, was die Grundproblematik der Metadatenbeschaffung unterstreicht, wurde anhand der Stichprobe bestätigt (siehe Kapitel 7.4, Abbildung 7.6).

Die Annahme, dass sich die Filmbewertungen je nach Plattform stark unterscheiden, konnte nicht direkt bestätigt werden (siehe 7.5, Abbildung 7.8). Hier wurde ein Test mit Metadaten der drei API–Provider durchgeführt, welcher zeigt, dass die Bewertung bei allen drei Providern im Schnitt fast identisch ist. Lediglich die Verteilung variiert hier leicht, es ist jedoch bei allen drei Anbietern eine ähnliche Verteilung zu beobachten (siehe Kapitel 7.5, Abbildung 7.8).

9.3 Aktuelle Probleme

Bei den Auswertungen und nochmaligem Reflektieren der verwendeten Algorithmen wurden Probleme aufgedeckt, die zum aktuellen Zeitpunkt des *libhugin*–Prototyps nicht bekannt waren.

Die problematische OFDb–Provider API, welche bereits während der Entwicklung auf einen damals allen Anschein nach funktionierenden Mirror zugegriffen hat, macht weiterhin Probleme. Hier zeigt das Erheben der Testmetadaten mit der *libhugin–harvest*–Bibliothek, dass das fehlerhafte Verhalten weiterhin besteht (siehe Kapitel 7.4, Abbildung 7.6). Hier werden Filme häufig ohne Inhaltsbeschreibung zurückgegeben. Des Weiteren wurde festgestellt, dass die API je nach Tageszeit und Serverauslastung, im Vergleich zu den anderen Providern instabil ist (siehe Kapitel 6.1, Abbildung 6.2).

Geschwindigkeitstests der *libhugin–harvest*–Bibliothek haben gezeigt, dass es hier bei den Providern ohne API Performanceunterschiede zu den Providern mit API gibt (siehe Kapitel 6.3, Abbildung 6.5, Abbildung 6.6). Als Grund wird hier das im Vergleich zum API–Provider aufwendigere Parsen der kompletten HTTP–Response vermutet. Hier wird aktuell die `BeautifulSoup`–Bibliothek verwendet. Eine Änderung des internen Parsers hat die Performance weiterhin verschlechtert. Hier wäre es wünschenswert, andere Ansätze zu finden, die diesen Vorgang performanter ausführen.

Weiterhin hat sich gezeigt, dass hier bei zwei Providern die Metadaten in keinem einheitlichen Encoding zurückgeliefert werden. Hier gab es Probleme mit den Umlauten beim Genre „Komödie”.

9.4 Ausblick

Zusammengefasst kann gesagt werden, dass mit dem *libhugin*–Prototyp das angesetzte Vorhaben, eine andere Herangehensweise beim Beschaffen der Metadaten im Vergleich zu den bisherigen Tools gut umgesetzt wurde. Aktuell gibt es jedoch noch vereinzelt Probleme, wie beispielsweise das oben genannte Problem mit dem Encoding oder auch die Geschwindigkeitseinbußen bei der Nutzung eines Providers ohne API.

Wie bereits in der Zusammenfassung der Projekarbeit (siehe [1], 8 Zusammenfassung) zur Implementierung der Bibliothek erwähnt, wäre es laut Autor sinnvoll, die Bibliothek weiter zu „verschlanken". Hier wird aktuell die Idee verfolgt, die „zweigeteilte" Bibliothek aus dem *libhugin–harvest* und *libhugin-analyze* Teil komplett separat zu entwickeln.

Generell sollten in Zukunft mehrere Provider implementiert werden, um die bisherigen Erkenntnisse mit einem größeren Onlinequellenspektrum zu bestätigen. Hier sollte bei weiteren Tests neben deutschsprachigen auch mehr Wert auf fremdsprachige Metadatenquellen gelegt werden.

A | Anhang A (GIL Limitierung)

Das folgende Code-Snippet wurde verwendet, um die GIL-Limitierung bei CPU-abhängigen Aufgaben zu messen:

```
#!/usr/bin/env python
# encoding: utf-8

import matplotlib.pyplot as plt
import numpy, time
from threading import Thread

def countdown(n):
    while n > 0: n -= 1

def plot(times):
    threads = [x for x, _ in times]
    values = [x for _, x in times]
    y_pos = numpy.arange(len(threads))
    plt.barh(y_pos, values, align='center', alpha=0.7, color='g')
    plt.yticks(y_pos, threads)
    plt.xlim(0, 30)
    plt.xlabel('time in seconds')
    plt.ylabel('number of threads')
    plt.title('python threaded cpu bound limitation because of the GIL.')
    plt.grid(True)
    plt.show()

if __name__ == '__main__':
    CNT = 100000000
    times = []
    for thread_cnt in range(0, 9, 2):
        thread_cnt = max(1, thread_cnt)
        cnt = range(1, thread_cnt + 1)
        threads = [Thread(target=countdown, args=(CNT // thread_cnt, )) for _ in cnt]
        for thread in threads:
            thread.start()
        start = time.time()
        for thread in threads:
            thread.join()
        times.append((thread_cnt, time.time() - start))
    plot(sorted(times))
```

B | Anhang B (Httplib Benchmark)

Das folgende Code-Snippet wurde zum Benchmarken der unterschiedlichen Python HTTP-Bibliotheken verwendet.

```
#!/usr/bin/env python
# encoding: utf-8

import concurrent.futures
from collections import defaultdict
from statistics import mean
import urllib.request
import time
import urllib3
import httplib2
import pylab

URLS = [
    'http://www.zeit.de', 'http://www.heise.de', 'http://www.golem.de',
    'http://www.krawall.de', 'http://www.phoronix.com', 'http://www.spiegel.de',
    'http://www.zeit.de', 'http://www.faz.de', 'http://www.focus.de',
    'http://www.filmstarts.de', 'http://www.moviepilot.de',
    'http://www.imdb.com', 'http://www.themoviedb.org', 'http://www.debian.org',
    'http://www.freebsd.org/de/'
] # 15 URLS

MAX_THREADS = 10

def fetch_urllib(url, timeout):
    return urllib.request.urlopen(url, timeout=timeout).readall()

def fetch_httplib2(url, timeout):
    h, c = httplib2.Http().request(url)
    return c

PM = urllib3.PoolManager()

def fetch_urllib3(url, timeout):
    return PM.request(url=url, method='GET').data

FUNCS = {
    'urllib': fetch_urllib,
```

```
    'httplib2': fetch_httplib2,
    'urllib3': fetch_urllib3
}

def download(threads=1, func=None):
    start = int(round(time.time() * 1000))

    with concurrent.futures.ThreadPoolExecutor(max_workers=threads) as executor:
        future_to_url = {executor.submit(func, url, 60): url for url in URLS}
        for future in concurrent.futures.as_completed(future_to_url):
            try:
                future.result()
            except Exception:
                pass

    end = int(round(time.time() * 1000))
    result = end - start
    return result

def plot(results):
    for lib in FUNCS.keys():
        t = [x[0] for x in results[lib]]
        s = [x[1] for x in results[lib]]
        pylab.plot(t, s, 'o-', label=lib)

    pylab.xlim(1, MAX_THREADS)
    pylab.xlabel('number of download threads')
    pylab.ylabel('time in milliseconds')
    pylab.title('performance scaling multithreaded download')
    pylab.grid(True)
    pylab.legend()
    pylab.show()

if __name__ == '__main__':
    N = 3
    results = defaultdict(dict)

    for name, func in FUNCS.items():
        run_results = []
        for threads in range(0, MAX_THREADS + 1, 2):
            threads = max(1, threads)
            avg_time = mean(download(threads=threads, func=func) for _ in range(N))
            run_results.append((threads, avg_time))
        results[name] = run_results

    plot(results)
```

C | Anhang C (Analyse Zeichenkettenvergleich)

Das folgende Code-Snippet wurde zum Benchmarken von unterschiedlichen Zeichenkettenvergleichsalgorithmen verwendet.

```
#!/usr/bin/env python
# encoding: utf-8

import matplotlib.pyplot as plt
import timeit

FUNCS = [{
    'func': '1 - normalized_damerau_levenshtein_distance("{s1}", "{s2}")',
    'fimport': 'from pyxdameraulevenshtein import normalized_damerau_levenshtein_distance',
    'label': 'damerau levenshtein'
}, {
    'func': 'difflib.SequenceMatcher(None, "{s1}", "{s2}", autojunk=False).ratio()',
    'fimport': 'import difflib',
    'label': 'ratcliff obershelp'
}, {
    'func': '1 - distance.nlevenshtein("{s1}", "{s2}")',
    'fimport': 'import distance',
    'label': 'levenshtein'
}
, {
    'func': 'adj_dlevenshtein.string_similarity_ratio("{s1}", "{s2}")',
    'fimport': 'import adj_dlevenshtein',
    'label': 'adjusted damerau levenshtein'
}
]

def benchmark(s1, s2, n, **kwargs):
    return timeit.timeit(
        kwargs['func'].format(s1=s1, s2=s2), kwargs['fimport'], number=n
    )

def plot(N, STEP):
    plt.title('String compare algorithm performance comparison')
    plt.ylabel('time in milliseconds')
    plt.xlabel('string length multiplication factor')
```

```
    plt.xlim(1, N / STEP)

    # plt.xscale('log')
    plt.yscale('log')

    plt.axes().xaxis.set_ticks_position('bottom')
    plt.axes().yaxis.set_ticks_position('left')
    plt.grid(True)

    plt.legend()
    plt.show()

def add_plot(plt, data, label):
    plt.plot(
        [y for y, _ in data],
        [x for _, x in data],
        'o-',
        label=label
    )

if __name__ == "__main__":
    s1 = 'Erdmännchen'
    s2 = 'Khaleesi'
    data = []
    N, STEP = 100, 5

    for algo in FUNCS:
        print('Benchmarking {algo}'.format(algo=algo['label']))
        for num in range(1, N + 1, STEP):
            fac = num // STEP + 1
            print(num, fac, STEP)
            timing = benchmark(s1 * fac, s2 * fac, 50, **algo)
            data.append((fac, timing * 100))
        add_plot(plt, data, algo['label'])
        data = []

    plot(N, STEP)
```

D | Anhang D (Rating Differenz)

Das folgende Code-Snippet wurde verwendet um Vergleichswerte zwischen Damerau-Levenshtein und Ratcliff-Obershelp zu ermitteln.

```
#!/usr/bin/env python
# encoding: utf-8

import difflib
from itertools import combinations_with_replacement
from adj_dlevenshtein import string_similarity_ratio
from pyxdameraulevenshtein import normalized_damerau_levenshtein_distance as norm_dl


if __name__ == "__main__":

    titles = ['Spiderman', 'Superman', 'Batman', 'Iron-Man']

    for str1, str2 in combinations_with_replacement(titles, 2):
        diff_lib = round(difflib.SequenceMatcher(None, str1, str2).ratio(),2)
        damerau_levenshtein = round(1 - norm_dl(str1, str2), 2)
        print(
            "{} <--> {}\ndifflib\t\t {}\ndlevenshtein\t {}\n".format(
                str1, str2, diff_lib, damerau_levenshtein)
        )
```

E | Anhang E (Gewichtetes Rating)

Das folgende Code-Snippet wurde zur gewichteten Rating-Ermittlung verwendet:

```
#!/usr/bin/env python
# encoding: utf-8

from adj_dlevenshtein import string_similarity_ratio
from pprint import pprint

def compare(search_title, titles, max_years=15):
    ratings = {}
    for title in titles:
        t, y = title.split(';')
        st, sy = search_title.split(';')
        year_sim = 1 - min(1, abs(int(y) - int(sy)) / max_years)
        rating = round((string_similarity_ratio(t, st) * 3 + year_sim) / 4, 3)
        ratings[title] = rating
    return ratings

def compare_dl(search_title, titles):
    s = string_similarity_ratio
    return {title: round(s(search_title, title), 3) for title in titles}

if __name__ == '__main__':
    a = 'Matrix; 1999'
    b = [
        'Matrix; 1999', 'Matrix; 2000', 'Matrix; 2001', 'Matrix; 1997',
        'Matrix, The; 1999', 'The Matrix; 2013', 'The East; 1999'
    ]

    for func in (compare, compare_dl):
        ratings = func(a, b)
        ratings = sorted(ratings.items(), key=lambda x: x[1], reverse=True)
        pprint(ratings)
```

F | Anhang F (Timeoutverhalten)

Das folgende Code–Snippet wurde für den Timeout–Test verwendet:

```
#!/usr/bin/env python
# encoding: utf-8

from hugin.harvest.session import Session
from statistics import mean
from collections import Counter
import sys

if __name__ == "__main__":
    with open(sys.argv[1], 'r') as fd:
        movies = fd.read().splitlines()

    cnt = Counter()
    sess = Session()
    providers = [
        'ofdbmovie', 'omdbmovie', 'tmdbmovie', 'videobustermovie', 'filmstartsmovie'
    ]

    N = 100

    for provider in providers:
        retries = []
        for movie in movies[:N]:
            title, year, imdbid = movie.split(';')
            r = sess.submit(
                sess.create_query(
                    title=title, year=year, cache=False, imdbid=imdbid,
                    providers=[provider], amount=1, retries=150
                )
            )
            retries.append(r[0]._retries)

        cnt[provider] = (min(retries), mean(retries), max(retries))
        print(cnt)
```

G | Anhang G (Antwortzeiten Http Metadatenquellen)

Das folgende Code-Snippet wurde zum Benchmarken von unterschiedlichen Antwortzeiten der Online-Plattformen verwendet.

```
#!/usr/bin/env python
# encoding: utf-8

from collections import defaultdict
from statistics import mean
import timeit
import numpy
import time
import matplotlib.pyplot as plt

ofdb = {
    'func': 'httplib2.Http().request("http://ofdbgw.geeksphere.de/search_json/{title}")',
    'fimport': 'import httplib2',
    'label': 'ofdb'
}

omdb = {
    'func': 'httplib2.Http().request("http://www.omdbapi.com/?s={title}")',
    'fimport': 'import httplib2',
    'label': 'omdb'
}

filmstarts = {
    'func': 'httplib2.Http().request("http://www.filmstarts.de/suche/?q={title}")',
    'fimport': 'import httplib2',
    'label': 'filmstarts'

}

videobuster = {
    'func': 'httplib2.Http().request("http://www.videobuster.de/' \
            'titlesearch.php?tab_search_content=movies' \
            '&view=title_list_view_option_list&search_title={title}")',
    'fimport': 'import httplib2',
    'label': 'videobuster'
```

```
}

tmdb = {
    'func': 'httplib2.Http().request("http://api.themoviedb.org/3/search/' \
            'movie?api_key=ff9d65f1e39e8a239124b7e098001a57&query={title}")',
    'fimport': 'import httplib2',
    'label': 'tmdb'
}

def benchmark(string, **kwargs):
    kwargs['func'] = kwargs['func'].format(title=string)
    return timeit.timeit(kwargs['func'], kwargs['fimport'], number=1)

def plot(providers):
    plt.yticks(y_pos, providers)
    plt.xlabel('time in milliseconds')
    plt.title('response time by online source.')
    plt.grid(True)
    plt.legend()
    plt.show()

if __name__ == "__main__":
    N = 10
    movies = [
        'Prometheus', 'Avatar', 'Matrix' #, 'Shame', 'Juno',
        #'Hulk', 'Rio', 'Alien', 'Wrong', 'Drive',
    ]

    times = defaultdict(list)

    for _ in range(N):
        for title in movies:
            for algo in [ofdb, tmdb, videobuster, omdb, filmstarts]:
                result = benchmark(title, **algo)
                times[algo['label']].append(result * 1000)

    for k, v in times.items():
        times[k] = (min(v), mean(v), max(v))

    providers = list(times.keys())
    y_pos = numpy.arange(len(times))

    for value, color, label in [(2, 'r', 'max'), (1, 'y', 'avg'), (0, 'g', 'min')]:
        response_ms = [p[value] for p in times.values()]
        plt.barh(y_pos, response_ms, align='center', alpha=0.7, color=color, label=label)

    plot(providers)
```

H | Anhang H (Antwortzeiten Libhugin Metadatenquellen)

Das folgende Code–Snippet wurde zum Benchmarken der Abfragegeschwindigkeit von *libhugin* verwendet.

```
#!/usr/bin/env python
# encoding: utf-8

from collections import defaultdict
from statistics import mean
from functools import partial
from hugin.harvest.session import Session
import timeit, numpy
import matplotlib.pyplot as plt

def benchmark(s, q):
    return s.submit(q)

def plot(times):
    providers = list(times.keys())
    y_pos = numpy.arange(len(providers))

    for value, color in [(2, 'r'), (1, 'y'), (0, 'g')]:
        response_ms = [p[value] for p in times.values()]
        plt.barh(y_pos, response_ms, align='center', alpha=0.7, color=color)

    plt.yticks(y_pos, providers)
    plt.xlabel('time in milliseconds')
    plt.title('libhugin download performance by online source.')
    plt.grid(True)
    plt.show()

if __name__ == "__main__":
    do_cache = False
    movies = [
        'Prometheus', 'Avatar', 'Matrix', 'Shame', 'Juno', 'Drive',
        'Hulk', 'Rio', 'Alien', 'Wrong'
    ]

    providers = [
```

```
        'ofdbmovie', 'tmdbmovie', 'videobustermovie', 'omdbmovie', 'filmstartsmovie'
    ]

    N = 3

    s = Session(parallel_downloads_per_job=1, cache_path='.')
    times = defaultdict(list)

    for _ in range(N):
        for title in movies:
            for provider in providers:
                qry = s.create_query(
                    title=title, cache=do_cache, providers=[provider],
                    amount=1, retries=150
                )
                result = timeit.Timer(partial(benchmark, s, qry)).timeit(number=1)
                times[provider].append(result * 1000)

    for key, value in times.items():
        times[key] = (min(value), mean(value), max(value))

    plot(times)
```

I | Anhang I (Libhugin Threaded Downloadgeschwindigkeit)

Das folgende Code-Snippet wurde für den Threaded-Search-Benchmark verwendet:

```
#!/usr/bin/env python
# encoding: utf-8

from collections import defaultdict
import timeit
import matplotlib.pyplot as plt
from functools import partial
from hugin.harvest.session import Session

N_THREADS = 20
N_MOVIES = 20

def benchmark(s, q):
    return s.submit(q)

def run(label, providers):
    times = defaultdict(list)
    for thread in range(1, N_THREADS + 1):
        s = Session(parallel_downloads_per_job=thread)
        q = s.create_query(
            title='Sin', cache=False, amount=N_MOVIES, retries=5,
            strategy='flat', providers=providers
        )
        result = timeit.Timer(partial(benchmark, s, q)).timeit(number=5)
        times[thread].append(result)

    plt.plot(
        [x[0] for x in times.items()],
        [x[1] for x in times.items()],
        label=label
    )

def plot():
    plt.xlim(1, N_THREADS)
    plt.xlabel('number of download threads', fontsize=14)
    plt.ylabel('time in seconds', fontsize=14)
```

```
    plt.title('libhugin threaded download comparsion', fontsize=14)
    plt.grid(True)
    plt.legend()
    plt.show()

if __name__ == "__main__":
    config = {
        'api': ['tmdbmovie', 'ofdbmovie', 'omdbmovie'],
        'no api': ['videobustermovie', 'filmstartsmovie'],
        'api + no api': [
            'tmdbmovie', 'ofdbmovie', 'omdbmovie',
            'videobustermovie', 'filmstartsmovie'
        ]
    }

    for label, providers in config.items():
        run(label, providers)

    plot()
```

J | Anhang J (IMDB Title Lookup)

Das folgende Code–Snippet wurde zum Beschaffen der Metadaten verwendet:

```
#!/usr/bin/env python
# encoding: utf-8

from hugin.harvest.session import Session
from collections import Counter
import re, json, httplib2, sys, os
import concurrent.futures

FOLDER = 'METADATA'
PROVIDERS = ['omdb', 'videobuster','filmstarts', 'tmdb', 'ofdb']

def read_file(filename):
    with open(filename, 'r') as f:
        return f.read().splitlines()

def fetch_data(
        session=None, title=None, year=None,
        src=None, mid=None, folder=None, source=None
    ):
    q = session.create_query(
        title=title.replace('|', '/'), year=year, imdbid=mid, providers=[src],
        amount=1, retries=150, language='de', cache=True, remove_invalid=True
    )
    return session.submit(q)

def get_movie_title(movieid):
    url =  'http://www.imdb.com/title/{imdb_id}'
    headers = {'Accept-Language': 'de'}
    h, c = httplib2.Http().request(url.format(imdb_id=movieid), headers=headers)
    regex = 'itemprop="name".*?>(.*?)</span>.*?\(.*?(\d{4}).*?\)'
    title, year = re.search(regex, c.decode().replace('\n', ' '), flags=re.S).groups()
    return '{};{};{}'.format(title, year, movieid)

def file_exists(folder, source):
    title, year, _ = folder.split(';')
    fmt = '{}/{}/{};{};{}.json'.format(FOLDER, folder, source, title, year)
    return os.path.isfile(fmt)
```

```
def create_folder_from_id(movieids):
    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
        future_to_title = {executor.submit(get_movie_title, mid) : mid for mid in movieids}
        for future in concurrent.futures.as_completed(future_to_title):
            title = future_to_title[future]
            try:
                data = future.result()
                os.mkdir("{F}/{SF}".format(F=FOLDER, SF=data.replace('/', '|')))
            except Exception as exc:
                print('%r generated an exception: %s' % (title, exc))

def trigger_download(s):
    results = Counter()
    for source in PROVIDERS:
        for num, folder in enumerate(folderlist):
            t, y, mid = folder.split(';')
            if not file_exists(folder, source):
                r = fetch_data(
                    session=s, title=t, src='{}movie'.format(source), mid=mid,
                    year=y, folder=folder, source=source
                )
                if r:
                    name = '{}/{}/{};{};{}.json'.format(FOLDER, folder, source, t, y)
                    with open(name, 'w') as f:
                        movie, *_ = r
                        f.write(json.dumps(movie.result_dict))
                else:
                    results[source] += 1
                    print(r, source, t)
    return results

if __name__ == '__main__':
    if sys.argv[1] == 'fetch':
        folders = os.listdir(FOLDER)
        folderlist = [
             i for i in folders if os.path.isdir('{F}/{SF}'.format(F=FOLDER, SF=i))
        ]
        s = Session(
            parallel_jobs=1, parallel_downloads_per_job=2,
            timeout_sec=20, cache_path='.'
        )
        print(trigger_download(s))
    else:
        movieids = read_file(sys.argv[1])
        if not os.path.exists(FOLDER):
            os.mkdir(FOLDER)
        create_folder_from_id(movieids)
```

K | Anhang K (Genre Analyse)

Das folgende Code-Snippet wurde für die Analyse der Genreinformationen verwendet:

```
#!/usr/bin/env python
# encoding: utf-8

from collections import Counter
from statistics import mean
from utils import analyze_folder
import pprint, os, sys

PROVIDERS = ['tmdb', 'ofdb', 'omdb', 'videobuster', 'filmstarts']

def json_iterator():
    for folder in os.listdir(sys.argv[1]):
        if not os.path.isdir(folder): continue

        providers = analyze_folder(folder)
        for provider, json_file in providers.items():
            yield provider, json_file

def count_attribute(attribute):
    results = {provider: Counter() for provider in PROVIDERS}
    for provider, json_file in json_iterator():
        for attr in json_file[attribute] or ['Kein Genre']:
            results[provider][attr] += 1
    return results

def count_attribute_len(attribute):
    results = {provider: [] for provider in PROVIDERS}
    for provider, json_file in json_iterator():
        results[provider].append(len(json_file[attribute] or []))
    return results

if __name__ == '__main__':
    counts = count_attribute('genre')
    pprint.pprint(counts)
    counts = count_attribute_len('genre')
    for prov, gen in counts.items():
        print('{}: ({}/{}/{})'.format(prov, min(gen), round(mean(gen), 2), max(gen)))
        for i in range(8): print('#{}: {}'.format(i, gen.count(i)))
```

L | Anhang L (Differenz Erscheinungsjahr)

Das folgende Code-Snippet wurde für die Analyse der Erscheinungsjahr-Differenzen verwendet:

```
#!/usr/bin/env python
# encoding: utf-8

from collections import Counter
from utils import analyze_folder
import difflib
import pprint
import os
import sys
import json

COUNTER = 0

def directors_equal(tmdb, nontmdb, threshold=0.95):
    d1, d2 = tmdb.get('directors', []), nontmdb.get('directors', [])
    if not all((d1, d2)):
        return False

    if len(d1) == len(d2) == 1:
        ratio = difflib.SequenceMatcher(None, d1[0].lower(), d2[0].lower()).ratio()
        return ratio > threshold

def test_print(tmdb, nontmdb):
    global COUNTER
    print("{})\n{}\n{}\n{}\n{}\n".format(
        COUNTER, tmdb['title'], nontmdb['title'],
        tmdb['directors'], nontmdb['directors']
    ))
    COUNTER += 1

def check_similarity(tmdb, nontmdb, threshold=0.90):
    diff = abs(int(tmdb.get('year')) - int(nontmdb.get('year')))
    ratio = difflib.SequenceMatcher(
        None, tmdb['title'].lower(), nontmdb['title'].lower()
    ).ratio()
```

```
    # handle movies by imdbid
    if nontmdb.get('imdbid') and tmdb.get('imdbid') == nontmdb.get('imdbid'):
        return diff

    # handle movies with no imdbid
    elif ratio > threshold and diff > 4 and not directors_equal(tmdb, nontmdb):
        test_print(tmdb, nontmdb)
        return diff

def calculate_year_diff():
    results = {p: Counter() for p in ['ofdb', 'omdb', 'videobuster', 'filmstarts']}
    for folder in os.listdir(sys.argv[1]):
        if not os.path.isdir(folder):
            continue

        providers = analyze_folder(folder)
        prov = {k: v for k, v in providers.items() if v.get('year')}
        if 'tmdb' not in prov:
            continue

        for name, jsonfile in prov.items():
            if name == 'tmdb':
                continue

            result = check_similarity(prov['tmdb'], jsonfile)
            if result is not None:
                results[name][result] += 1

    return results

if __name__ == '__main__':
    pprint.pprint(calculate_year_diff())
```

M | Anhang M (Unvollständigkeit Metadaten)

Das folgende Code-Snippet wurde für die Analyse der Unvollständigkeit der Metadaten verwendet:

```
#!/usr/bin/env python
# encoding: utf-8

from collections import Counter
from utils import analyze_folder
import pprint, sys, os, json

PROVIDERS = {
    'tmdb':2500, 'ofdb':2500, 'omdb':2500, 'videobuster':2444, 'filmstarts':2427
}
def update_value_availability(results, provider):
    name, jsonfile = provider
    for key, value in jsonfile.items():
        if value:
            results[name][key] += 1

def analyze():
    results = {p: Counter() for p in PROVIDERS.keys()}
    for folder in os.listdir(sys.argv[1]):
        if os.path.isdir(folder):
            providers = analyze_folder(folder)
            for provider in providers.items():
                update_value_availability(results, provider)
    return results

def invert_results(data):
    for name, values in data.items():
        for key, cnt in values.items():
            data[name][key] = PROVIDERS[name] - cnt
    return data

if __name__ == '__main__':
    data = analyze()
    pprint.pprint(invert_results(data))
    print()
```

N | Anhang N (Ratingverteilung)

Das folgende Code-Snippet wurde für die Analyse der Ratingverteilung in der Stichprobe verwendet:

```
#!/usr/bin/env python
# encoding: utf-8

import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
from utils import analyze_folder
from statistics import mean
import pprint
import sys
import os
import json

PROVIDERS = ['tmdb', 'ofdb', 'omdb']

def analyze():
    results = {p: [] for p in PROVIDERS}
    for folder in filter(os.path.isdir, os.listdir(sys.argv[1])):
        if folder == '__pycache__':
            continue
        for name, metadata in analyze_folder(folder).items():
            if name not in PROVIDERS:
                continue

            rating = metadata.get('rating')
            if rating:
                results[name].append(float(rating))

    return results

def calculate_min_mean_max(data):
    rating_stats = {name: Counter() for name in PROVIDERS}
    for name, ratings in data.items():
        rating_stats[name] = (min(ratings), round(mean(ratings), 2), max(ratings))
    return rating_stats

def count_ratings(data):
```

```
    rating_counts = {name: Counter() for name in PROVIDERS}
    for name, metadata in data.items():
        for value in metadata:
            rating_counts[name][value] += 1
    return rating_counts

def plot_distribution(data):
    colors = ['r', 'g', 'b']
    _, axes = plt.subplots(len(data), sharex=True, sharey=True)
    axes = list(axes)

    axes[0].set_ylabel('Number of given ratings')
    # axes[0].set_xlabel('Rating')

    for idx, provider in enumerate(data.keys()):
        ax = axes.pop()
        ratings = Counter()
        for i in range(10 + 1):
            ratings[i] = 0

        for rating, times in data[provider].items():
            ratings[round(rating * 2) / 2] += times

        X = np.array(list(ratings.keys()))
        Y = np.array(list(ratings.values()))

        ax.bar(X, Y, 0.4, color=colors.pop())
        ax.set_title('Rating Distribution of "{p}"'.format(p=provider))
        ax.set_xticks(X)
        ax.grid(True)

    plt.show()

if __name__ == '__main__':
    data = analyze()
    plot_distribution(count_ratings(data))
    pprint.pprint(calculate_min_mean_max(data))
```

O | Anhang O (Utilities)

Die Funktion im folgenden Code-Snippet wird von Skripten zum Einlesen der Metadaten verwendet:

```
#!/usr/bin/env python
# encoding: utf-8

import os
import json

def analyze_folder(path):
    data = {}
    for f in os.listdir(path):
        with open(os.path.join(path, f), 'r') as fp:
            provider, _, _ = f.split(';')
            data[provider] = json.loads(fp.read())
    return data
```

P | Literaturverzeichnis

[1] Christoph Piechula. *Design und Implementierung eines modularen Filmmetadaten Such- und Analysesystems*. Hof University, 2014.

[2] M. Lutz. *Programming Python*. O'Reilly Media, 2010.

[3] JW Ratcliff and DE Metzener. Pattern matching: the gestalt approach. *Dr. Dobb's Journal*, 1988. URL: http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/1988/8807/8807c/8807c.htm.

[4] S. Cordts. *Datenqualität in Datenbanken*. Mana-Buch, 2012.

[5] Gonzalo Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 2001.

[6] M.J. Atallah and M. Blanton. *Algorithms and Theory of Computation Handbook, Second Edition: General Concepts and Techniques*. Chapman & Hall/CRC Applied Algorithms and Data Structures series. Taylor & Francis, 2010.

[7] P. Christen. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Data-Centric Systems and Applications. Springer, 2012.

[8] Benno Stein. *Automatische Extraktion von Schlüsselwörtern aus Text*. Bauhaus–Universität Weimar, 2006.

[9] Stuart Rose, Dave Engel, Nick Cramer, and Wendy Cowley. Automatic keyword extraction from individual documents. *Text Mining*, pages 1–20, 2010.

[10] Michael W Berry and Jacob Kogan. *Text mining: applications and theory*. John Wiley & Sons, 2010.

[11] Christopher Pahl. *Algorithmik und Evaluation des Musikempfehlungssystems Libmunin*. Hof University of Applied Sciences, 2014.

[12] Jeffrey EF Friedl. *Reguläre Ausdrücke*. O'Reilly Germany, 2009.

[Link–1] http://www.imdb.com. [Stand: 3.8.2014].

[Link–2] http://www.xbmc.org. [Stand: 3.8.2014].

[Link–3] http://windows.microsoft.com/de-de/windows7/products/features/windows-media-center. [Stand: 3.8.2014].

[Link–4] http://wiki.xbmc.org/index.php?title=NFO_files/movies. [Stand: 3.8.2014].

[Link–5] http://dvdxml.com/p/faq/faq.php?0.cat.2.3. [Stand: 3.8.2014].

[Link–6] http://www.mediaelch.de. [Stand: 3.8.2014].

[Link–7] http://libhugin.readthedocs.org/en/latest/developer_api/api.html#developing-a-content-provider-plugin. [Stand: 3.8.2014].

[Link–8] http://libhugin.rtfd.org. [Stand: 3.8.2014].

[Link–9] https://developers.google.com/api-client-library/python/guide/thread_safety. [Stand: 3.8.2014].

[Link–10] http://de.wikipedia.org/wiki/Filmgenre. [Stand: 3.8.2014].

[Link–11] http://www.ofdbgw.org/. [Stand: 3.8.2014].

[Link–12] http://www.crummy.com/software/BeautifulSoup/bs4/doc/. [Stand: 3.8.2014].

[Link–13] http://up.nullcat.de/ae2849906cdd47d843eea40381a086e4/metadata.tar.bz2. [Stand: 3.8.2014].

[Link–14] http://matplotlib.org. [Stand: 3.8.2014].

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Hof, den 3. August 2014 *Christoph Piechula*