Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

Wintersemester 2015/ 2016

Fakultät für Mathematik, Informatik und Naturwissenschaften

64-194 Projekt Parallelrechnerevaluation

Dr. Michael Kuhn, Dr. Julian Kunkel, Konstantinos Chasapis

Plagiatserkennung in Dokumentensammlungen

Dokumentation

Christoph Liegmann (Matr.-Nr.: 6567995)

Kevin Thien (Matr.-Nr.: 6523467)

Jiyan Jonsdotter (Matr.-Nr.: 6087314)

Inhalt

1. Einleitung

1.1. Handlungs-/ Forschungsfeld

Dieser Dokumentation liegen ein absolviertes Projekt und ein angefertigtes Software-Programm zugrunde. Das Projekt *Parallelrechnerevaluation* des Fachbereiches Informatik der Universität Hamburg ist als Modul in mehreren Studiengängen enthalten und behandelt verschiedene Aspekte der an Bedeutung gewinnenden Parallelrechner. So können beispielsweise Dateisysteme und die Speicherung von großen Datensätzen, aber auch das Laufzeitverhalten von Hard- und Softwareeinheiten betrachtet werden. Vor dem wissenschaftlichen Hintergrund werden jeweils Evaluationen durchgeführt um letztlich fundierte Aussagen über die behandelten Themen der Parallelrechner treffen zu können[1].

Unsere Arbeitsgruppe besteht aus drei Studierenden, von denen sich Zwei im Bachelorstudium und Einer im Masterstudium befinden. Zugeteilt haben wir uns, im Rahmen des Projekts, für das Thema „Plagiatserkennung in Dokumentensammlungen“.

Durch drei Zwischenbesprechungen und einer Zwischenpräsentation während des laufenden Semesters erhielten wir regelmäßige Rückmeldungen bezüglich unserer aktuellen Arbeit und konnten dadurch sehr zielorientiert das Projekt (weiter-)entwickeln.

1.2. Aufgabenstellung

Unser Thema ist, wie bereits genannt, die „Plagiatserkennung in Dokumentensammlungen“. Dieses etwas offenere Thema stellte uns zunächst vor die Frage, an welchem Punkt wir ansetzen sollten. Nach einer Absprache hatten wir jedoch eine konkrete Vorstellung, was die Aufgabenstellung anging: Es galt eine Plagiatssoftware zu schreiben, bei der wir eigenständig verschiedene Algorithmen zur Plagiatserkennung herausfinden und implementieren sollten. Als zweiten Schritt sollten wir das Laufzeitverhalten verschiedener Algorithmen testen und analysieren, sodass wir schlussendlich aus einer Laufzeit-Algorithmus-Übersicht in Form eines Diagramms alle relevanten Informationen entnehmen können.

Diese Dokumentation soll darüber hinaus als eine Art Protokoll unseres Vorgehens dienen und softwarebegleitend unsere Entscheidungen und Implementierungen erklären.

Geschrieben wurde unsere Plagiatssoftware hauptsächlich in der Programmiersprache Java. Ein Teil, der im späteren Kapitel dieser Dokumentation näher erläutert wird, wurde in der neueren Sprache, dem *R-Code*, implementiert.

[1] Vgl. projektbegleitende Homepage *https://wr.informatik.uni-hamburg.de/teaching/wintersemester_2015_2016/parallelrechnerevaluation (Stand, 16.03.2016)*

2. Hauptteil

2.1. Realisierung

Das Ziel bestand darin, Plagiate durch Softwareeinsatz zu entdecken. Da im wissenschaftliche Bereich vorzugsweise mit PDF-Formaten (*Portable Document Format*) gearbeitet wird, haben wir uns dazu entschieden, dass unsere zu schreibende Software primär die Plagiate in PDF-Dateien finden soll.

Diesbezüglich hatten wir uns folgendes vereinfachtes Prinzip vorgestellt: Auf der einen Seite liegt uns eine Sammlung (Corpus) an vorhandenen PDF-Dateien zugrunde und auf der Anderen haben wir das einzelne Dokument, das auf Plagiate überprüft werden soll. Vor diesem Hintergrund wollten wir die Software konstruieren, wobei diese physikalische Ordnung der Dateien für uns im weiteren Verlauf eine wichtige Rolle spielt.

Eine weitere Frage, die für uns eine hohe Relevanz besaß, war, welche Form die Software am Ende des Projekts annehmen sollte. Was diesen Aspekt anging, waren wir uns alle einig, dass die Funktionalität des Programmes im Vordergrund stehen soll, weshalb wir uns gegen eine grafische Benutzeroberfläche entschieden haben. Somit würde eine gestaltete Oberfläche (GUI), die gegebenenfalls von dem eigentlichen Sinn des Projekts ablenkt wegfallen und der Fokus könnte auf den wichtigeren Dingen liegen, wie den Algorithmen, der Implementierung oder der genaueren Plagiatssuche.

Ziel einer jeden Plagiatssoftware ist es möglichst genau ein Plagiat ausfindig zu machen. Dazu dient in den meisten Fällen ein Wert, wie zum Beispiel eine „prozentuale Angabe über die Ähnlichkeit zwischen dem Analysedokument und den Inhalten anderer Dokumente“[2]. Einen Algorithmus zu finden und zu implementieren, der in den häufigsten Fällen einen hohen prozentualen Wert liefert, also sehr genau die Textpassagen analysiert, war unser Anspruch.

Grundsätzlich ist als Benutzer einer solchen Software auf folgendes zu achten: „Sogenannte Plagiatserkennungssoftware findet keine Plagiate, sondern nur identische Textstellen. Die endgültige Entscheidung darüber, ob ein Text ein Plagiat ist oder nicht, muss von der zuständigen Lehrkraft getroffen werden. Die Software sollte nur ein Hilfsmittel, aber kein Prüfstein sein. Blindes Vertrauen in automatisch generierte Plagiatsberichte ist unverantwortlich.“[3]

Bevor wir auf das Design eingehen, soll der Begriff des Plagiats geklärt werden, da dieser in der Literatur zum Teil unterschiedlich definiert ist, sodass Missverständnisse nicht ausgeschlossen werden können. Deshalb halten wir uns in diesem Bericht an die folgenden Definitionen:

---

[2] *Churer Schriften zur Informationswissenschaft, Wissensklau, Unvermögen oder Paradigmenwechsel?, Plagiate als Herausforderung für Lehre, Forschung und Bibliothek, 2009, S. 97*

[3] *http://plagiat.htw-berlin.de/software/2013-2/, (Stand 28.12.2015)*

Plagiate:

Unterschieden wird zwischen acht Plagiatsvarianten: Das Totalplagiat, das Übersetzungsplagiat, das Teilplagiat, das Ideenplagiat, das altruistische Plagiat, das Autoplagiat, das Verbal- und Bildplagiat. Relevant für uns sind das Totalplagiat, bei dem eine vollständige Textübernahme eines fremden Textes stattgefunden hat und das Teilplagiat, das am häufigsten vorkommt. Dabei werden fremde Textpassagen mit eigenen Gedanken kombiniert, ohne eine Quellenangabe anzugeben.

Schwierig nachzuweisen wäre noch das Ideenplagiat, bei dem ein kompletter Hintergedanke eines Textes übernommen wird, ohne dabei den Urheber beziehungsweise eine Quelle anzugeben[4].

2.1.1. Design

In dem folgenden Abschnitt möchten wir, beginnend mit der eigentlichen Idee, auf die genaue Entstehung der Software eingehen.

Zunächst haben wir uns überlegt, welches die Schlüsselaktivitäten unseres Programms werden sollen. Gestellte Fragen waren zum Beispiel: Was für Voraussetzungen müssen für einen Ablauf erfüllt sein, welchen Zustand müssen die Daten haben und wie hat das Ziel auszusehen?

Anhand der Fragen wurde uns schnell bewusst, dass sich die Datensammlung in einem Verzeichnis befinden und nicht verteilt in verschiedenen Laufwerken abgespeichert werden sollte. Somit ist die physikalische Sicherung der Daten für uns sehr wichtig. Neben dem haben wir beschlossen ausgehen von PDF- und Textdateien arbeiten zu wollen. Um aber nun die Texte dieser Dateien für unsere Algorithmen benutzen zu können, müssen wir jedes einzelne PDF-Dokument konvertieren, weshalb unser erstes Ziel darin bestand einen *Converter* zu implementieren.

Der *Converter* sollte letztlich so aufgebaut sein, dass er eine Eingabe in Form einer PDF-Datei in eine Liste umwandelt, die anschließend intern weiterverwendet wird. Somit wird die PDF-Datei extrahiert und keine neu generierte Datei in erzeugt.

Während des Konvertierens werden noch einige Merkmale der Texte abgeändert, sodass das Arbeiten mit den Algorithmen später leichter fällt: Die Buchstaben „ä“, „ö“, „ü“ und „ß“ werden jeweils in „ae“, „oe“, „ue“ und „ss“ umgewandelt. Außerdem wird durch den Befehl *Lower Case* der gesamte Text lediglich in Kleinbuchstaben betrachtet und verarbeitet. Zeilenumbrüche werden durch *System.getProperty(“line.seperator“)* speziell abgefangen und abgeändert, sodass kein Bindestrich in den Wörtern auftritt.

[4] *Vgl.: Krüger, A., Plagiate – Ein ständiges Problem an Universitäten, Studienarbeit, Norderstedt.2009, S. 6f.*

Durch das Vorliegen dieser konvertierten Listen sind zunächst alle Voraussetzungen erfüllt, um mit Hilfe von implementierten Algorithmen potenzielle Plagiate aufzudecken.

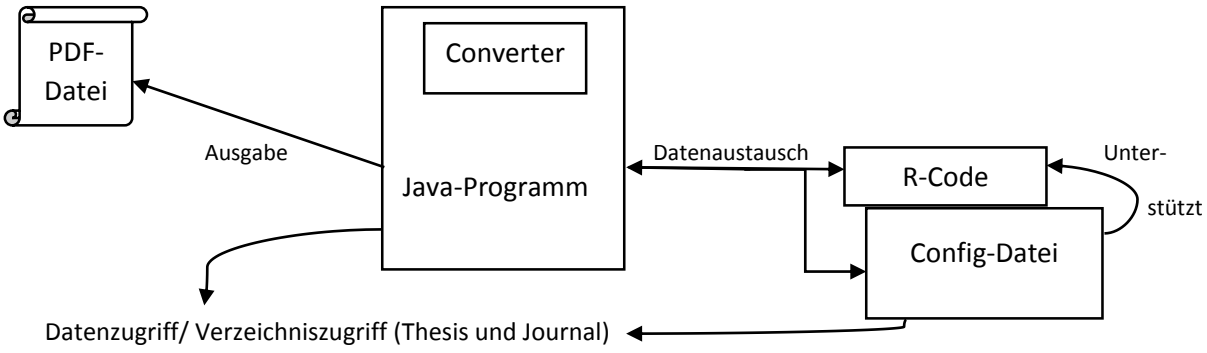
Obwohl in einem weiterführenden Kapitel gezielt auf das Verwenden und Implementieren mit der Programmiersprache R eingegangen wird, möchte ich dieses hier dennoch kurz nennen. Eine Teilaufgabe bestand darin, Algorithmen im R-Code zu implementieren. Weshalb sich gerade diese Sprache eignet und welche Vorteile sie bietet, wird später erläutert. Unser nächster Schritt bestand schließlich darin, dass wir einen R-Code schreiben, der ein Verfahren zur Plagiatserkennung selber beinhaltet und zum Teil aufruft. Hierbei handelt es sich um das Stemmen von Wörtern. Der Quelltext wird durch eine Konfigurationsdatei (kurz *Config*-Datei) unterstützt. Die erste Idee bestand darin, dass das R-Programm die Config-Datei aufruft und ausführt, in der weitere Verfahren zur Plagiatserkennung implementiert sind. Letztlich haben wir es allerdings so umgesetzt, dass der R-Code und die Konfigurationsdatei unabhängig voneinander eingesetzt werden können. Auf die jeweiligen Algorithmen wird ebenso noch eingegangen.

Die Ausgabe, die getätigt wird, ist das Wichtigste der gesamten Software. Sie liefert eine PDF-Datei, in der alle Dokumente inklusive ihrer Stellenangaben aufgelistet sind, die Übereinstimmungen, und somit unter Umständen Plagiat, aufweisen. Ein Überprüfen im Einzelfall ist absolut notwendig.

Die Konfigurationsdatei haben wir so aufgebaut, dass in ihr zwei Verzeichnisse eingegeben werden können. Zum einen ist es das Quellverzeichnis, in der die Datei liegt, die zu überprüfen ist (Thesis) und zum anderen ist es das Verzeichnis des Corpus‘, also der zugrunde liegenden Dokumentensammlung (Journal). Neben diesen Einstellungen befinden sich weitere Algorithmen zum Ermitteln von Plagiaten, wie zum Beispiel das Hashwertverfahren.

Zusammenfassend lässt sich sagen, dass unsere Software in drei Einheiten eingeteilt werden kann. Die jeweiligen Einheiten sind nach ihrer Funktion getrennt und sorgen somit für eine übersichtliche Handhabung und im späteren Verlauf auch Wartung. Es sind:

1. Der Converter (Zusammenhängend mit dem Java-Programm)
2. Das R-Programm
3. Die Config-Datei mit Verzeichnispfaden und Algorithmen



Der Converter greift zunächst auf die Thesis zu, um die genannten Schritte zu durchlaufen. Daher erfolgt an dieser Stelle ein Datenzugriff. Die interne Liste wird entweder an den R-Code oder die Config-Datei weitergeleitet. Hier erfolgt die Verarbeitung durch die Algorithmen mit Hilfe der Verzeichniseinstellungen in der Konfigurationsdatei. Ein Ergebnis wird per Datenaustausch dem eigentlichen Java-Programm übermittelt, der anschließend eine Ausgabe in Form einer PDF-Datei generiert.

2.1.2. Implementierung

Während der Implementierung kam uns der Gedanke eine Art Regler zu verwenden, mit dessen Hilfe die Genauigkeit der arbeitenden Algorithmen eingestellt werden kann. Diesen „Threshold“ implementierten wir aber erst, nachdem die Algorithmen geschrieben waren. Zusätzlich haben wir einen Zähler implementiert, den wir *Min-Counter* genannt haben. Durch ihn kann eingestellt werden, wie viele Wörter eines Satzes mindestens übereinstimmen müssen.

Config-Datei

Die Config-Datei, die dem Projekt beigeführt ist, ist das Schlüsselelement zur Verwendung des Programms. In Dieser sind verschiedene Informationen, die vom Nutzer manipuliert werden können, um verschiedene Ergebnisse zu erzielen. Dabei sind bestimmte Vorgaben einzuhalten.

In der ersten Zeile befindet sich der Name des Ordners, der sich im selben Verzeichnis wie das Projekt befindet, in dem sich die gesamten Dokumente befinden, mit dem sich das neue Dokument vergleichen soll.
Eine Zeile darunter befindet sich der Name des Ordners, der sich ebenso im selben Verzeichnis wie das Projekt befindet, in dem sich das Dokument befindet, welches überprüft werden soll.
Die nächsten zwei Werte beziehen sich auf den Hashwertalgorithmus. Je nach Wert unterscheiden sich die Ergebnisse beim ausführen des Algorithmus (siehe Leistungsanalyse).
Die letzten beiden sind notwendig, um den R-Code ausführen zu lassen. Zuerst wird aufgegriffen, ob es sich beim Vergleichen, um deutsche Texte (welche als „de“ in der Datei gekennzeichnet wird) oder um Englische Texte („en“). Der letzte Punkt in der Config-Datei dient zur Benutzung des R-Codes. Wenn in der Config-Datei ein „true“ zu lesen ist, wird der R-Code benutzt, sonst wird dieser vermieden.

Verwendung

Bei der Verwendung des Programms wird hauptsächlich auf die Config-Datei zugegriffen. Wenn das Programm startet, wird diese Datei im Hauptverzeichnis gesucht. Dort muss sich auch das „words.txt“ Dokument befinden, welches die Wörter beinhaltet, die später gefiltert werden. Die Unterordner, die untersucht werden sollen, sollten sich ebenso in

diesem Verzeichnis befinden, damit beim Suchen der Dateien keine Fehler entstehen. Der Rest geschieht beim Ausführen automatisch. Das Ergebnis der Untersuchung wird ins Verzeichnis mit den gesamten Dokumenten abgelegt, mit der Betitelung „Ergebnis vom DATUM, UHRZEIT“. Diese sollte dann in ein separates Verzeichnis gelegt werden, bevor dieses Dokument als Vergleichsdokument dient.
Bevor das Programm ausgeführt wird, sollte man in der Config-Datei kontrollieren, ob alle Einstellungen korrekt getroffen wurden, damit die Kontrolle sich auch als möglichst sinnvoll gestaltet.

Filter

Beim Konvertieren der PDFs, kann durch eine gezielte Filterung von Wörtern dafür gesorgt werden, dass eine geringere Anzahl an Wörtern betrachtet werden muss, wodurch die Datensätze geringer ausfallen. Bevor es zum eigentlichen Filtern kommt, muss man sich aber noch bestimmte Gedanken machen.
Was für Wörter möchte man Filtern? In erster Linie Präpositionen, sowie jede Art von Artikel, Partikel und Adverbien (zum kleinen Teil auch Adjektive), da diese nur zur Umschreibung des Objektes und der Lage dienen.
Präpositionen beschreiben das Verhältnis verschiedener Objekte miteinander. Dabei dienen sie nur zur Umschreibung, was im Kontext nicht ausschlaggebend ist. Die Artikel sind nur Begleiter die einem Objekt zugeordnet werden. Bestimmte Artikel können auch als Relativpronomen vorkommen. Da aber ebenso Kommas gefiltert werden, kann diese Art der Verknüpfung vernachlässigt werden und so auch herausgefiltert werden.
Verben und Nomen sollten jedoch weiter vorhanden bleiben, weil diese im Kontext dafür sorgen, dass eine Handlung beziehungsweise eine Situation (Szenario, Beispiel, …) beschreiben.

Ausgabe der Dateien

Nach dem Ausführen der Algorithmen müssen die Ergebnisse, die beim Überprüfen, ob ein Plagiat vorliegt, noch erfasst werden. Es bieten sich dazu verschiedene Möglichkeiten an. Um möglichst lange die Ergebnisse zu sichern, sollten diese als Dokument erfasst werden, dann abgespeichert werden, sodass darauf immer wieder zugegriffen werden kann. Als gängigstes Format erweist sich das PDF, da diese vom Programm ausgelesen werden. Das Erstellen eines PDFs ist dementsprechend nicht aufwendig. In diesem PDF werden nun die Ergebnisse des Algorithmus erfasst. Je nach Algorithmus spielen da andere Vorgehensweisen beziehungsweise andere Informationen eine Rolle.
Beim Hashverfahren sollten die Textausschnitte, die eine Ähnlichkeit haben rausgesucht und mit dazugehörigem Dokument notiert werden, damit diese Passage nachgelesen werden kann.
Bei der Wortliste sollten zu den Dokumenten notiert werden, wie viel Wörter im zu vergleichenden Text vorhanden sind, damit man ungefähr abschätzen kann, wie umfangreich die Überschneidung ist. Dazu schreibt man die Anzahl der überschneidenden Wörter auch noch auf. Die Abweichung der überschneidenden Wörter, also die, die sich

überschneiden, aber unterschiedlich oft vorkommen, spielt dabei ebenso eine Rolle. Wenn gleiche Stilmittel verwendet werden, können die gleichen Worte beziehungsweise Formulierungen benutzt werden. Dadurch überschneiden sich die Worte. Jedoch muss man beachten, dass bestimmte Worte, in einer speziellen Thematik, unvermeidlich sind. Daher sollte man ebenso darauf achten, inwiefern die Häufigkeit der überschneidenden Worte, über und unterschritten wird. Um dann dies Präzise darzustellen eignen sich prozentuale Angaben zu den Werten. Die Anzahl der überschneidenden Wörtern, sowie die Häufigkeitstendenz sind ideal zur Bestimmung, ob ein Plagiat vorliegt und sollten daher als prozentualer Wert im Dokument vermerkt werden.

2.1.3. R-Code

Die Programmiersprache R ist eine 1993 entstandene funktionale und objektorientierte Sprache zur Arbeit mit statistischen Werten. Sie ist eine General Purpose Programming Language um effiziente Programme schreiben zu können, beziehungsweise um effizient Berechnungen durchführen zu können. Diese Berechnungen können über die Kommandozeile entweder direkt, oder über das Ausführen von Dateien mit einem Programmcode als Inhalt, ausgeführt werden. Ein relevanter Aspekt der Sprache R ist das Operieren mit stochastischen Daten. Es lassen sich auf hoher Abstraktionsebene Funktionen schreiben, was der Behandlung von stochastischen Einsatzbereichen zugutekommt. Durch online erweiterbare Pakete besitzt R viele Schnittstellen zu anderen Programmiersprachen und stellt ohne aufwendige Implementierungen Algorithmen und Funktionen für den Nutzer zur Verfügung. Schwierig ist dabei nur bereits existierende Pakete zu finden.

Für unser Projekt der Plagiatserkennung ist R ein hilfreiches Mittel, da es eine Schnittstelle zu der Programmiersprache Java bietet, in der ein weiterer Teil unseres Software-Projektes geschrieben ist, und darüber hinaus hilfreiche Algorithmen in Form von Paketen leicht einbezogen und verwendet werden können.

So ist unter anderem das Verfahren des Wörter Stemmens verfügbar. Der Algorithmus wird in einem späteren Teil detaillierter beschrieben.

Nachfolgend wird anhand von Beispielen die Funktionalität der Programmiersprache R verdeutlicht. Die Beispiele haben keinen Bezug zu unserer Projektarbeit.

Es kann sich die folgende Eingabe der Sprache R in die Konsole ergeben:

```
> a = c(1,2,3)

> b = c(1,2,3)

> a+b

[1] 2 4 6

> a+runif(b)
```

[1] 1.361338 2.151640 3.687127

Dabei werden zwei Vektoren a und b erschaffen, welche miteinander addiert werden. Es sind beliebige Operationen, wie zum Beispiel Multiplikationen, auf Diesen möglich. In unserem Beispiel verwenden wir die Funktion runif(), welche einen zufälligen Wert zwischen 0 und 1 zurückgibt und der die Normalverteilung zugrunde liegt (unif = "uniform").

Es gibt zudem die Funktionen dnorm() und pnorm(). Beiden liegt die Gauß'sche Normalverteilung zugrunde. Dabei berechnet dnorm() zu seinem Parameter den Wert der Gauß'schen Glockenkurve, während pnorm() das Integral berechnet. Es gelten also die folgenden Beziehungen:

dnorm(-x) = dnorm(x)

pnorm(x) + pnorm(-x) = 1

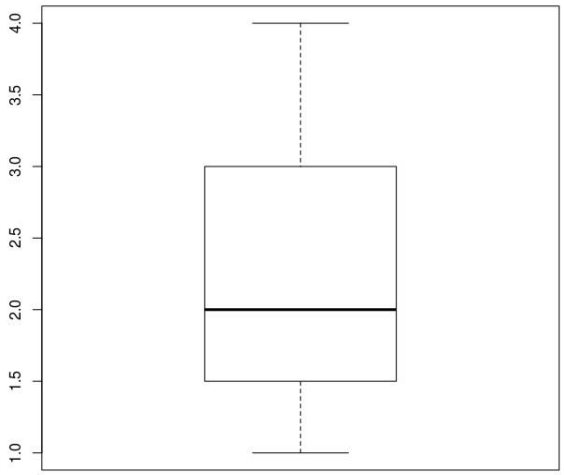
pnorm(0) = 0.5

Die Gauß'sche Glockenkurve ist symmetrisch, weshalb die erste Gleichung gilt. Zudem ist das Gesamtintegral gleich eins, was die zweite Gleichung erklärt. Die dritte Gleichung folgt direkt aus der Ersten.

Auch die Binomialverteilungen finden Anwendung durch Funktionen wie *dbinom(a,20,0.1)*. Der Wert 20 entspräche hierbei der Anzahl der Ereignisse und 0.1 entspräche der Wahrscheinlichkeit für den Erfolg eines Ereignisses.

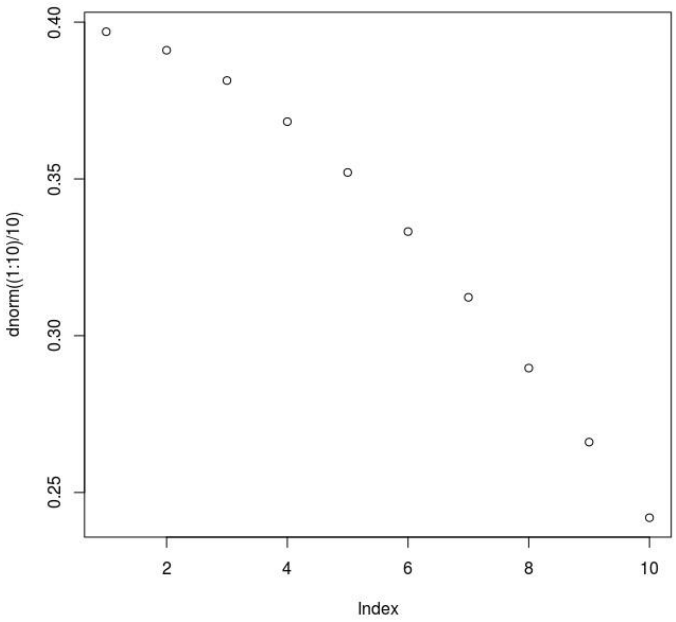
Alle diese Operationen lassen sich beliebig auf Vektoren oder Matrizen anwenden. Die Funktion summary() liefert eine Zusammenfassung der wichtigsten Fakten über diese Daten.

```
> summary(c(1,2,2,4))
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    1.75    2.00    2.25    2.50    4.00
>boxplot(c(1,2,2,4))
```



(Abbildung 1)

Die Ergebnisse von Daten lassen sich in verschiedenen gängigen Plots darstellen. Eine bekannte Darstellung ist beispielsweise *Boxplot* (siehe Abbildung 1), welches alle Werte, die bei summary() aufgelistet werden, optisch veranschaulicht (siehe Abbildung 2). Weiterführend wären Histogramme und das Aufzeichnen von Werten möglich.



(Abbildung 2)[5]

[5] https://www.r-project.org/ (Stand, 28.03.2016)

Alle bisherigen Befehle wurden direkt in die Konsole eingegeben. Durch den Befehl source(), lassen sich Dateien als Programmcode ausführen. Diese können Berechnungen anstellen und diese durch Plots oder den Befehl cat() ausgeben. Wie bereits erwähnt ist R turingmächtig, weil es Kontrollstrukturen wie die if-Anweisung und Schleifen wie die while-Schleife oder gar die foreach-Schleife kennt.

```
> x = 3

> if(x==3)cat("hallo")

hallo> if(x==2)cat("hallo") # das Hallo entstammt der letzten Zeile

> if(x==2)cat("hallo")

> while(x > 0)x=x-2

> cat(x)
```

Eine besondere Stellung nimmt hierbei die foreach-Schleife ein. Diese ist bekannt aus anderen Programmiersprachen wie Java oder C++ (entweder wenn man die neuesten Standards betrachtet, wo die erweiterte for-Schleife direkt enthalten ist, oder die foreach-Schleife der STL-Library, welche über Iteratoren über mengenartigen Objekten iterieren kann). Wie in anderen Sprachen auch, lässt sich so effizient über Mengen operieren.

```
> foreach(i=1:10) %do%{i**2}

[[1]]

[1] 1


[[2]]

[1] 4


[[3]]

[1] 9


[[4]]
```

[1] 16

[[5]]

[1] 25

[[6]]

[1] 36

…

Es wird bei foreach-Schleifen generell über Mengen iteriert. In jeder Iteration wird der entsprechende Wert automatisch in eckigen Klammer geschrieben und der Befehl wird ausgeführt. Der Befehl i**2 berechnet die Quadratwurzel eines Wertes. Ähnliches ergeben die abstrakteren Operationen auf Vektoren:

*> c(1,2,3)**2*

[1] 1 4 9

*> c(1,2,3)**3*

[1] 1 8 27

Damit die foreach-Schleife in ihrer einfachsten Form benutzt werden kann, muss zuvor das entsprechende Package installiert sein. In dem Fall von foreach wäre das also der Befehl *install.packages("foreach")*. Anschließend muss über den Befehl *library(foreach)* die foreach-library für uns sichtbar gemacht werden. Allgemein lassen sich so alle möglichen Packages einbinden und das Schema zum Einbinden des foreach-packages *(install.packages("") - library())* lässt sich auf viele andere Packages übertragen.

Die foreach-Schleife benutzt den sogenannten Infix Operator, welcher analog ist zur Operatorenüberladung. Diese ist beispielsweise bekannt aus C++, wo der Befehl für eine Klasse statt +(a,b) mit a+b geschrieben werden kann, was intuitiver und besser lesbar ist. In Java ist dieser Umstand zwar auch aber weniger bekannt, die Klasse String kann über den +-Operatore verknüpft werden. Der Infix-Operator in R ist jedoch flexibler und es lassen sich beliebige Operatoren-Konstrukte basteln.

`%add%`<-function(x,y) x+y

1 %add% 2 # gives back 3

Diesen Umstand macht sich die foreach-Schleife zu Nutze, um das Backend für entsprechende Operationen zu definieren. Diese kann entweder normal sequentiell Befehle abarbeiten oder mittels der Packages: doMC, doSNOW, doMPI mit entsprechender Parallelität arbeiten. Zudem ist es möglich mit Zufallszahlen zu arbeiten. Lässt man die Frage beiseite, in wie fern durch den Computer erzeugte Zahlen überhaupt zufällig sein können, ergibt es Sinn in reproduzierbare und nicht reproduzierbare Zufallszahlen zu unterschieden. Prinzipiell hängt die Zufallszahlengenerierung von dem Zustand des Algorithmus‘ zur Zufallszahlenerzeugung ab. Nutzt man beispielsweise zur Zufallszahlenerzeugung 6 * 4 Bytes (ein Maschinenwort hat hier 4 Bytes), so entspricht dies 6*4*8 Bit. Damit kann der Algorithmus also maximal 2^192 verschiedene Zustände annehmen, wonach der Algorithmus wieder alte Zustandsmuster erzeugen muss.

Es kann erwünscht sein Zufallszahlen bei Simulationen reproduzierbar zu haben (um Ergebnisse nachzuvollziehen oder zu Debugging Zwecken). Im Kontext der Parallelität heißt dies, dass für jeden Prozess ein eigener Zufallszahlen-Stream angelegt wird, damit diese sich nicht in die Quere kommen können. Das Package doRNG liefert ein Backend für die foreach-Schleife und erzeugt reproduzierbare Zufallszahlen.

```
set.seed(200)
foreach(i=1:3) %dopar% {runif()}
set.seed(200)
foreach(i=1:3) %dopar% {runif()}
#won‘t be identical
```

Dieses Beispiel wird zu einer sehr hohen Wahrscheinlichkeit nicht identische Werte erzeugen.

```
set.seed(200)
foreach(i=1:3) %dorng% {runif()}
set.seed(200)
foreach(i=1:3) %dorng% {runif()}
#will be identical / reproducible
```

Durch das %dorng% hingegen werden beide Male sicher die gleichen Werte herauskommen.

Durch die Packages rmpi und pbdmpi unterstützt R MPI. OpenMP wird jedoch noch nicht unterstützt, ein entsprechendes Package dazu (romp) befindet sich in der

Erstellungsphase. Die Packages snow und snowfall lassen es zu von dem jeweiligen Parallelismus, beziehungsweise im Fall von snowfall eventuellen nicht-Parallelismus zu abstrahieren. Operationen werden also ausgeführt, aber es bleibt quasi dem Programmierer überlassen, ob er nur einen PC mit mehreren Kernen hat, ein Cluster oder nur einen sequentiellen Prozess hat. Programme werden durch die Apply Funktionen ausgeführt, wobei davon ausgegangen wird, dass die Daten seperierbar sind und gewisse Funktionen auf all diesen Daten ausgeführt werden kann. Es können also keine Nachrichten zwischen den Knoten hin und her geschickt werden, was snow und snowfall zwar weniger ausdrucksstark macht, allerdings lassen sich in den meisten Fällen wo die Daten seperierbar sind so effizient parallele Programme schreiben.

Packages stellen eine Möglichkeit dar R zu erweitern und werden von der R-Community entwickelt. Sie stellen das Gegenstück zu Librarys in anderen Programmiersprachen dar. Der zugrundeliegende Code ist zumeist in C geschrieben, um trotz der hohen Abstraktion und Effektivität von R eine hohe Performance zu erreichen.

Zusammenfassend lässt sich sagen, dass die Verwendung der Programmiersprache R für unser Projekt passend ist. Wie zuvor verdeutlicht handelt es sich um eine sehr analytische Sprache, was für uns bezüglich der späteren Laufzeitanalyse von Vorteil ist.

Shared	**Mixed**	**Distributed**
(romp)	snow	rmpi
	snowfall	pbdmpi
doMC	doSNOW	doMPI
	rlecuyer	
	doRNG	

(Abbildung 3)

2.2. Algorithmen

Es gibt verschiedene Verfahren zur Plagiatserkennung. Dabei unterscheiden sich die Algorithmen zum Teil sehr stark. Für unsere Plagiatssoftware haben wir gedacht, dass wir verschiedene dieser Verfahren implementieren und anhand unterschiedlicher Kriterien (Laufzeitverhalten und Genauigkeit) analysieren und bewerten. Unsere These war es zudem, dass für verschiedene Textarten (Wissenschaftliche Arbeiten, Protokolle, Präsentationen, Stichwortlisten) ein unterschiedliches Einsetzen von Algorithmen angepasstere Ergebnisse liefert. Inwiefern dies zutreffend ist, wird noch beschrieben werden.

Fokussiert haben wir uns bei Allem auf die nachfolgend genannten Algorithmen:

Fingerprint-Algorithmus:

Dieses Verfahren ist auch bekannt als der Algorithmus mit Hash-Funktion. Der Ansatz ist, dass nicht jeder einzelne Buchstabe verglichen werden muss, sondern Hashfunktionen sind üblicherweise so definiert, dass zusammengefasste Werte oder Elemente einer großen Menge auf sogenannte Hashwerte abgebildet werden. Demnach wird jeder Satz in kleinere Teile zerlegt, denen dann ein Hashwert zugewiesen wird. Diese Werte werden anschließend verglichen, sodass eine Aussage darüber getroffen werden kann, inwiefern eine Ähnlichkeit zwischen zwei Sätzen besteht.

In diesem Zusammenhang sollte man mit einem Hash-Set arbeiten. Ein Set ist so definiert wie eine Menge, in der keine Elemente doppelt vorkommen dürfen. Somit wird in unserem Zusammenhang ein Satz, trotz mehrfachen Vorkommens, nur einmal mit einem Hashwert versehen und abgespeichert.

String-Matching-Algorithmus:

Der String-Matching-Algorithmus wird auch als der naive Ansatz bezeichnet. Sowohl der zu überprüfende Text (das Pattern), als auch der Corpus liegen in Form von Zeichenketten (Strings) vor. Das Pattern wird nun indexweise mit dem eigentlichen Text verglichen und gegebenenfalls verschoben. Nach jedem Verschieben wird das Pattern erneut von Vorne verglichen[6].

Beispiel:

String A (Pattern): CDEF, String B (Corpus): ABCDEFG

String B: ABCDEFG

String A: CDEF	Indizes der Stelle 0 stimmen nicht überein, verschieben
CDEF	Indizes der Stelle 1 stimmen nicht überein, verschieben
CDEF	Indizes stimmen überein

Diese Version des Algorithmus‘ arbeitet allerdings nicht sehr effizient, da eine Worst-Case-Betrachtung eine enorm lange Laufzeit benötigt. Wie im folgenden Beispiel zu sehen ist, müssen alle möglichen Vergleiche stattfinden, bis der Algorithmus terminiert:

[6] Vgl.: Prof. Dr. Dahmen, W., RWTH Aachen Universität: Mathematisches Praktikum – SoSe 14, S.1

Beispiel:

String A (Pattern): AAAAB

String B (Corpus): AAAAAAAAA…

String B: AAAAAAAAAAA….

String A: AAAAB

AAAAB

AAAAB

….

Es liegt eine Laufzeit von O(n^2) vor, wobei durch Variation des Verfahrens ein weitaus besseres Laufzeitverhalten erzielt werden kann. So sind die Verfahren des Knuth-Morris-Pratt und des Boyer-Moore vor dem Hintergrund der Laufzeit effektiver. Dies liegt daran, dass durch eine Patternanalyse in einigen Fällen Vergleichsprozesse eingespart werden können.

Wortliste:

Hierbei handelt es sich um ein Verfahren, bei dem jedes einzelne Wort gezählt wird. Da ein Text allerdings zu einem Großteil aus Füllwörtern (wie *und*, *aber*, *ein*, *eine*, …) besteht, wollten wir diese zunächst herausfiltern, damit sie nicht mitgezählt werden. In diesem Zusammenhang entdeckten wir das Zipf's Law, das genau diesen Sachverhalt aufgreift. So erhält jedes einzelne Wort einen Ranking-Wert, der seine allgemeine Häufigkeit in Texten beschreibt.

$$c = f * r$$

Das Produkt aus der Frequenz des Wortes, also der Häufigkeit der Benutzung, und eines Ranking-Wertes ergibt die Konstante c.

Die ersten fünf Wörter, die deshalb in unserem Kontext irrelevant sind, werden zu Beginn des Algorithmus' identifiziert und anschließend ignoriert. Daraufhin wird eine Häufigkeitstabelle angelegt, die Auskunft darüber gibt, wie oft ein einzelnes Wort in dem Textabschnitt vorkommt. Die beiden entstehenden Listen (Häufigkeit der Wörter in dem Pattern und Häufigkeit der Wörter des Corpus) können nun unter der Benutzung der Gesamtanzahl der Worte einen prozentualen Wert erzeugen, der eine Aussage darüber trifft, inwiefern eine Übereinstimmung vorliegt.

Porter-Stemmer-Algorithmus:

Im Prinzip gibt es drei Arten von Stemmern. Die erste Variante von Stemmern sucht in Dictionaries nach den verschiedenen Wortstämmen und gleicht diese ab. Somit hat dieser Ansatz nur weniger Ergebnisse, welche false-positive sind (also bei denen eine Wortübereinstimmung errechnet wurde, aber eigentlich keine vorhanden ist). Zur Einschätzung der Leistung eines Algorithmus‘ werden oft die Begriffe true-positive, false-positive, true-negative und false-negative verwendet. Ein Ergebnis ist true-positive, wenn ein Algorithmus ein Ergebnis liefert und dieses auch vorhanden sein soll. Ein Ergebnis ist hingegen false-positive, wenn ein Ergebnis vorhanden ist, das jedoch eigentlich nicht in der Ergebnismenge sein sollte. Die Gegenstücke hierzu stellen true-negative und false-negative dar.

Ein weiterer Ansatz ist das Abgleichen über Algorithmen wie der Porter-Stemmer-Algorithmus, indem generisch aus Wörtern über Regeln die Wortstämme generiert werden.

Neben diesen Ansätzen gibt es noch den sogenannten Hybride Ansatz, welcher beide Ansätze kombiniert. Dazu wird in dem Dictionary nachgesehen ob das Wort existiert. Falls nicht wird durch die Algorithmen versucht das Wort zu stemmen und es wird erneut untersucht ob das Wort in dem Dictionary vorhanden ist. Das so durch den Algorithmus und das Dictionary entstandene Stemm-Wort wird dann als Ergebnis genommen. Es gilt, dass Dictionaries für alte Wörter besser funktionieren, während Algorithmen für neuere Wörter besser funktionieren. Die Ursache liegt wohl darin, dass neue Wörter und ihre Abwandlungen besser in bestimmte Regel-Schemata passen.

Grundsätzlich arbeiten all diese Algorithmen sehr eng an der Grammatik und stammen aus dem Bereiche der Linguistik. Es wird, analog zu den anderen Plagiaterkennungsverfahren, zunächst das Pattern, dann der Corpus betrachtet und im dritten Schritt Beide verglichen. Das Besondere dabei ist, dass die einzelnen Wörter der Textabschnitte auf ihren jeweiligen Wortstamm zurückgeführt werden, wodurch auch das Abändern eines Wortstammes als Plagiat aufgedeckt werden kann.

Im Vergleich kann man schließlich die vorhandenen Wortstämme des Corpus mit denen des Patterns gegenüberstellen und erhält, ähnlich zu dem Verfahren der Wortliste, einen prozentualen Wert über die Übereinstimmung.

Beispiel:

Die folgenden Wörter können alle auf den gleichen Wortstamm zurückgeführt werden:

Beeindrucken, beeindruckend, beeindruckender, beeindruckendsten → beeindruck

Eine Abwandlung des Stemmer-Algorithmus‘ ist der *N-Gramm Stemmer-Algorithmus.* Durch ihn wird jedes einzelne Wort eines Textes in ein Bigramm beziehungsweise in ein N-

Gramm zerlegt. Damit ist gemeint, dass die jeweils aufeinanderfolgenden Buchstaben eines Wortes notiert und anschließend gezählt werden. So besitzt jedes Wort eine Bigrammmenge mit verschiedenen enthaltenen Elementen. Dieses Verfahren wird sowohl für das Pattern, als auch für die Texte des Corpus‘ durchgeführt. Anschließend werden die Bigrammmengen des Patterns und des Corpus‘ betrachtet und eine gemeinsame Schnittmenge gebildet. Über eine Ähnlichkeitsformel kann die prozentuale Übereinstimmung errechnet werden[7]:

$$\text{Ähnlichkeit S} = \frac{2*(Bigramme\ Wort1 \cap Bigramme\ Wort2)}{Bigramme\ Wort1 + Bigramme\ Wort2}$$

Beispiel:

Wort 1: genießen, Wort 2: genießbar

Bigramm Wort 1: genießen → ge en ni ie eß ße en

Bigramm Wort 2: genießbar → ge en ni ie eß ßb ba ar

Bigrammmenge Wort 1: {ge, en, ni, ie, eß, ße} (6 Elemente)

Bigrammmenge Wort 2: {ge, en, ni, ie, eß, ßb, ba, ar} (8 Elemente)

Gemeinsame Bigrammmenge (Schnittmenge): {ge, en, ni, ie, eß} (5 Elemente)

Ähnlichkeitsberechnung: $S = \frac{2*5}{6+8} = \frac{10}{14} \approx 0{,}71$

Zwischen den Wörtern 1 und 2 liegt eine prozentuale Ähnlichkeit von ungefähr 71%.

[7] Vgl. Frakes, W., Information Retrieval: Data structures and algorithms, S. 136 f.

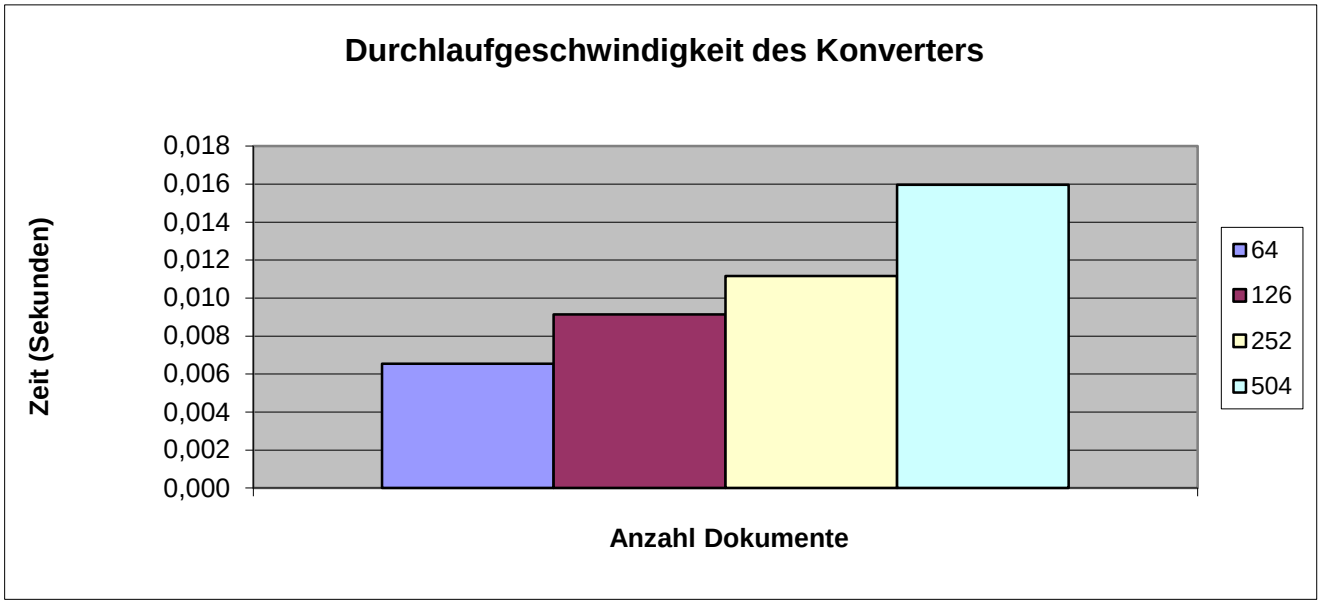
2.3. Leistungsanalyse

Die Leistungsanalyse wurde hinsichtlich zwei Aspekten durchgeführt. Die Verfahren zur Plagiatserkennung können anhand ihrer Schnelligkeit, also der Laufzeit charakterisiert werden, während es auch möglich ist einen Algorithmus hinsichtlich seiner Effizienz zu testen. Effizienz meint in unserem Falle das sinngemäße und erfolgreiche Ausführen einer Plagiatssuche. Auch in diesem Bereich kann es zu Unterschieden zwischen den Algorithmen kommen.

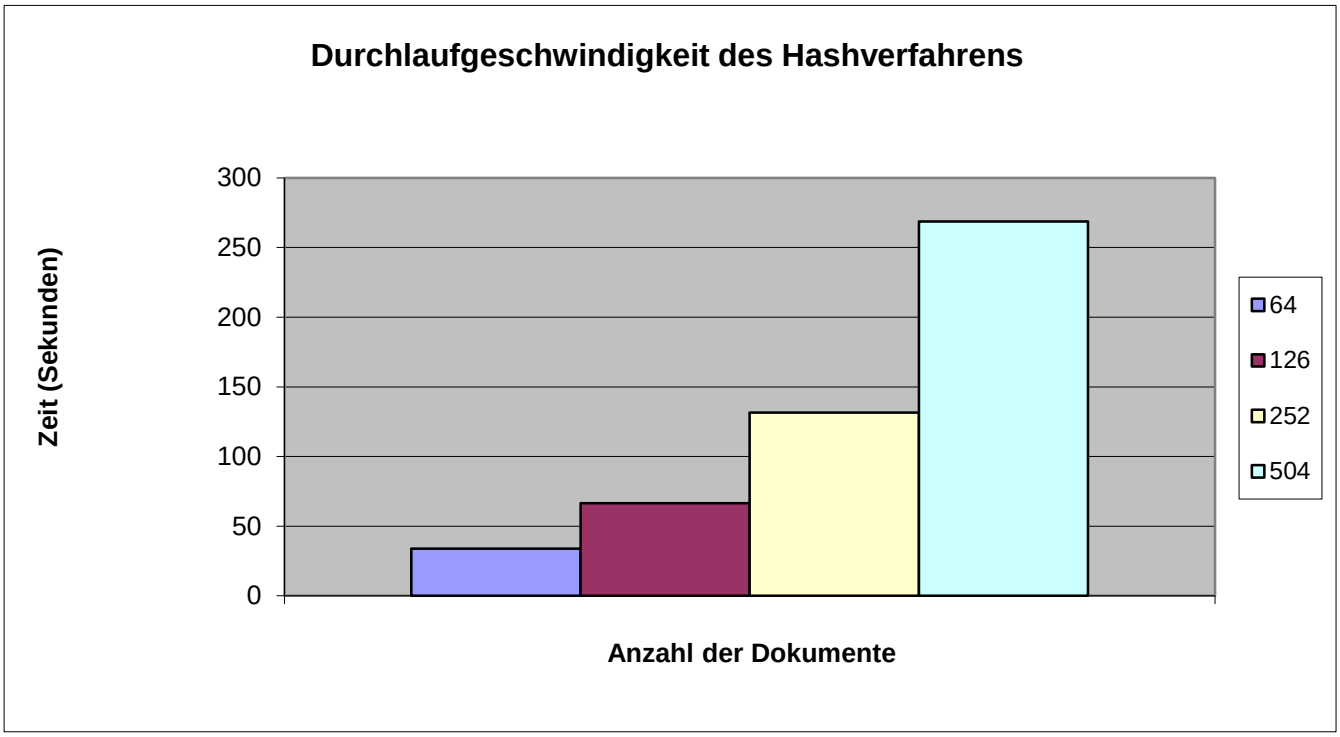
Beeinflusst wird die Analyse der Effizienz durch die Einstellungen des Thresholds (Schwellwert) und des Min-Counters. Setzt man bei einem Durchlauf den Wert des Thresholds, der in einem Wertebereich von 0.0 bis 1.0 liegt, zu niedrig, werden mehr Ergebnisse als potenzielle Plagiate ausgegeben, als wenn den Wert höher wählt. Der Min-Counter hat den Wertebereich eines Integerwertes, was also -2.147.483.648 bis 2.147.483.647 bedeutet. Es ergibt sich folgender Zusammenhang:

	Threshold		
Min-Counter		Niedrige Einstellung	Hohe Einstellung
	Niedrige Einstellung	Viele Plagiatsfunde (ungenau)	Sätze können kurz sein und müssen aber einen hohen Anteil an Gleichheit aufweisen
	Hohe Einstellung	Sätze müssen länger werden und nur eine geringe Gleichheit besitzen	Wenige Plagiatsfunde (genau)

Nachfolgend wird die Laufzeitanalyse durchgeführt. Um eine vergleichbare Aussage treffen zu können, waren die Einstellungen bei allen getesteten Algorithmen dieselben. Zuerst erfolgt die Analyse vor dem zeitlichen Hintergrund und anschließend vor dem Hintergrund der Effizienz.

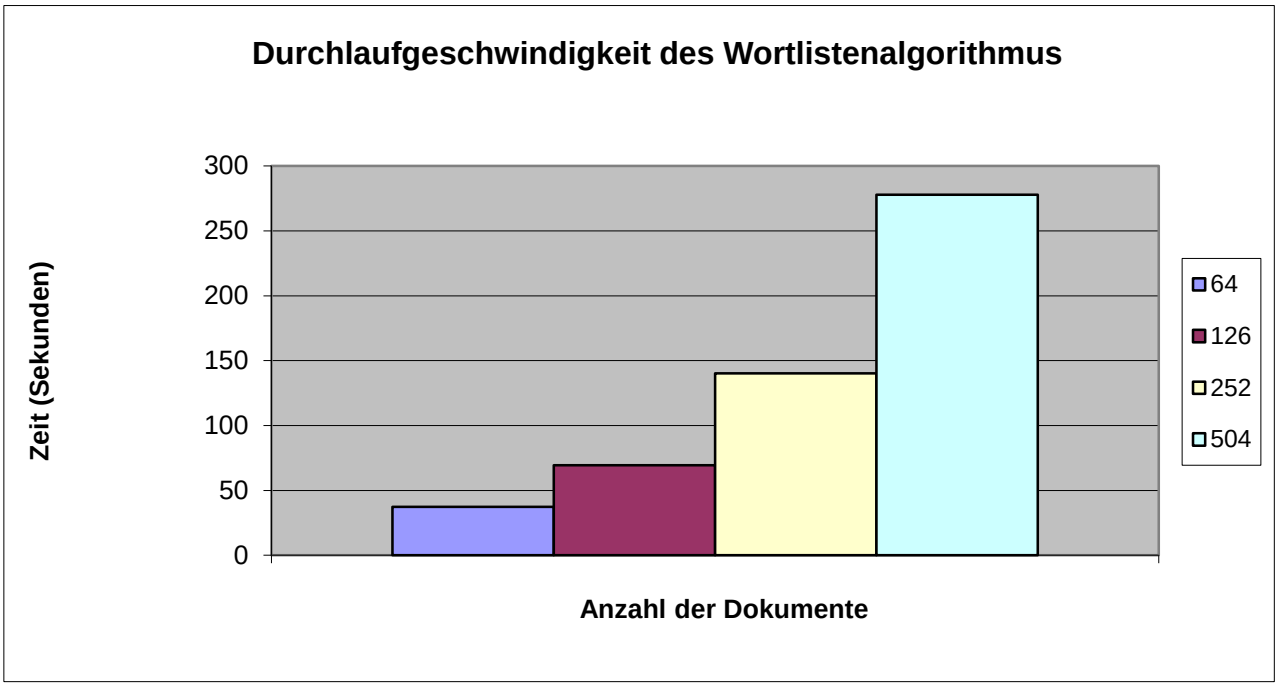


Das zuvor dargestellte Diagramm zeigt zunächst die Geschwindigkeit des Konvertierens verschiedener Dateiumfänge. Wie vermutet steigt die Konvertierungszeit mit größer werdender Anzahl von Dokumenten. Dennoch läuft dieser Prozess sehr schnell ab, da die Zeit für 500 Dokumente im Millisekunden-Bereich liegt.



Hier abgebildet ist die Durchlaufgeschwindigkeit des ersten Algorithmus‘. Verwendet wurde das Hashwertverfahren und in jeweils vier Fällen getestet. Je größer die Anzahl der Dokumente ist, desto länger benötig das Verfahren bis zum Terminieren. Bei knappen 500

Dokumenten benötigt der Algorithmus knappe fünf Minuten. Es lässt sich die These aufstellen, dass der Algorithmus für 100 Dokumente ungefähr 60 Sekunden braucht.



In dem oben stehenden Diagramm wird der zweite Algorithmus hinsichtlich seiner Laufzeit getestet. Es handelt sich um den Wortlistenalgorithmus. Auch in diesem Falle steigt die Dauer der Bearbeitung mit der Anzahl der Dokumente. Interessant ist, dass man bei 252 Dokumenten sehr gut erkennen kann, dass dieses Verfahren länger braucht, als das Hashwertverfahren. Insgesamt unterscheiden sich die beiden Algorithmen um wenige Sekunden.

Bei dem Leistungstest bezüglich der Effizienz wurde wie folgt vorgegangen: Zunächst überprüften wir zwei identische Dokumente miteinander, während wir in einem weiteren Schritt und den gleichen Einstellungen zwei komplett unterschiedliche Dokumente miteinander verglichen. Ziel war es bei dem ersten Test möglichst viele Plagiate zu ermitteln, wobei bei dem zweiten Durchlauf bestenfalls keine Übereinstimmungen festgestellt werden sollten. Verwendet wurde das Hashwertverfahren.

Unsere Tests ergaben folgende Ergebnisse:

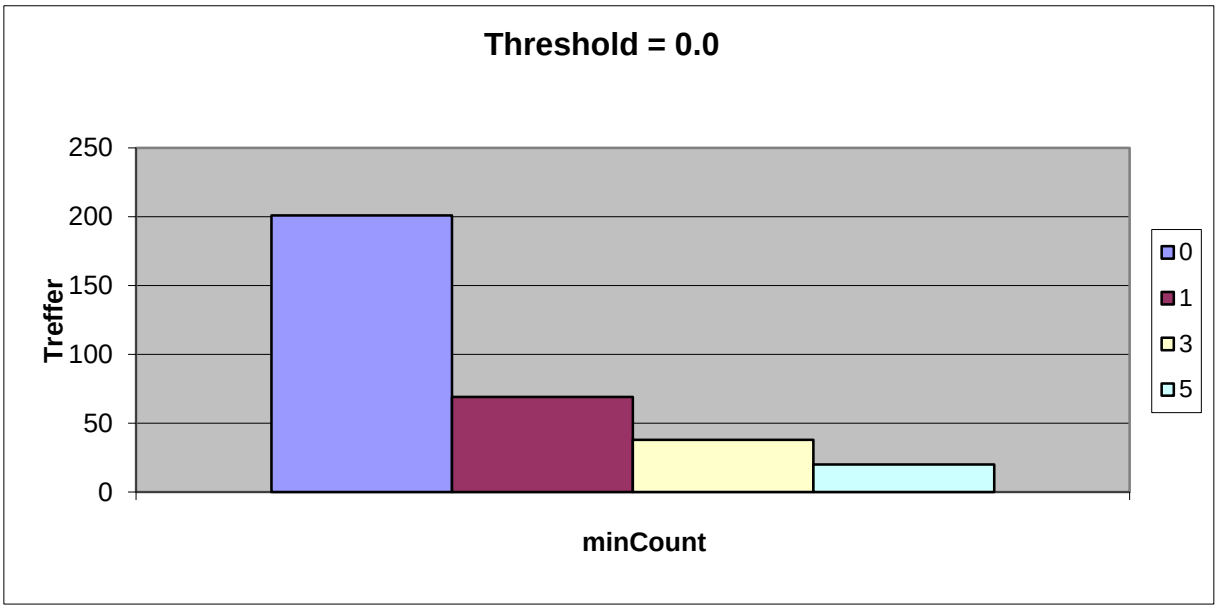
Bei identischen Dokumenten:

Threshold	Min-Count	Ergebnis	Wertung
0.0	0	201	Plagiat
0.0	1	69	Plagiat
0.0	3	38	Plagiat
0.0	5	20	Plagiat
0.0	9	3	Plagiat
0.1	0	167	Plagiat
0.1	1	69	Plagiat
0.1	3	38	Plagiat
0.1	5	20	Plagiat
0.1	9	3	Plagiat
0.5	0	61	Plagiat
0.5	1	49	Plagiat
0.5	3	38	Plagiat
0.5	5	20	Plagiat
0.5	9	3	Plagiat
0.9	0	61	Plagiat
0.9	1	49	Plagiat
0.9	3	38	Plagiat
0.9	5	20	Plagiat
0.9	9	3	Plagiat

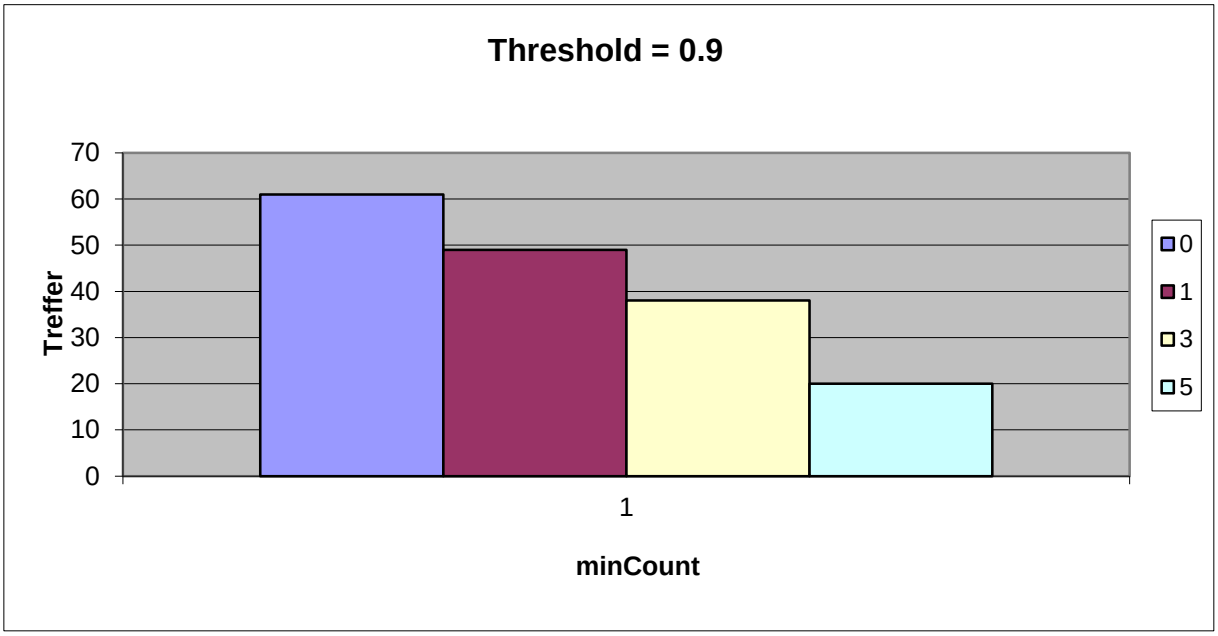
Bei unterschiedlichen Dokumenten:

Threshold	Min-Count	Ergebnis	Wertung
0.0	0	120	Plagiat
0.0	1	1	Plagiat
0.0	3	0	Echt
0.0	5	0	Echt
0.0	9	0	Echt
0.1	0	76	Plagiat
0.1	1	1	Plagiat
0.1	3	0	Echt
0.1	5	0	Echt
0.1	9	0	Echt
0.5	0	3	Plagiat
0.5	1	0	Echt
0.5	3	0	Echt
0.5	5	0	Echt
0.5	9	0	Echt
0.9	0	3	Plagiat
0.9	1	0	Echt
0.9	3	0	Echt
0.9	5	0	Echt
0.9	9	0	Echt

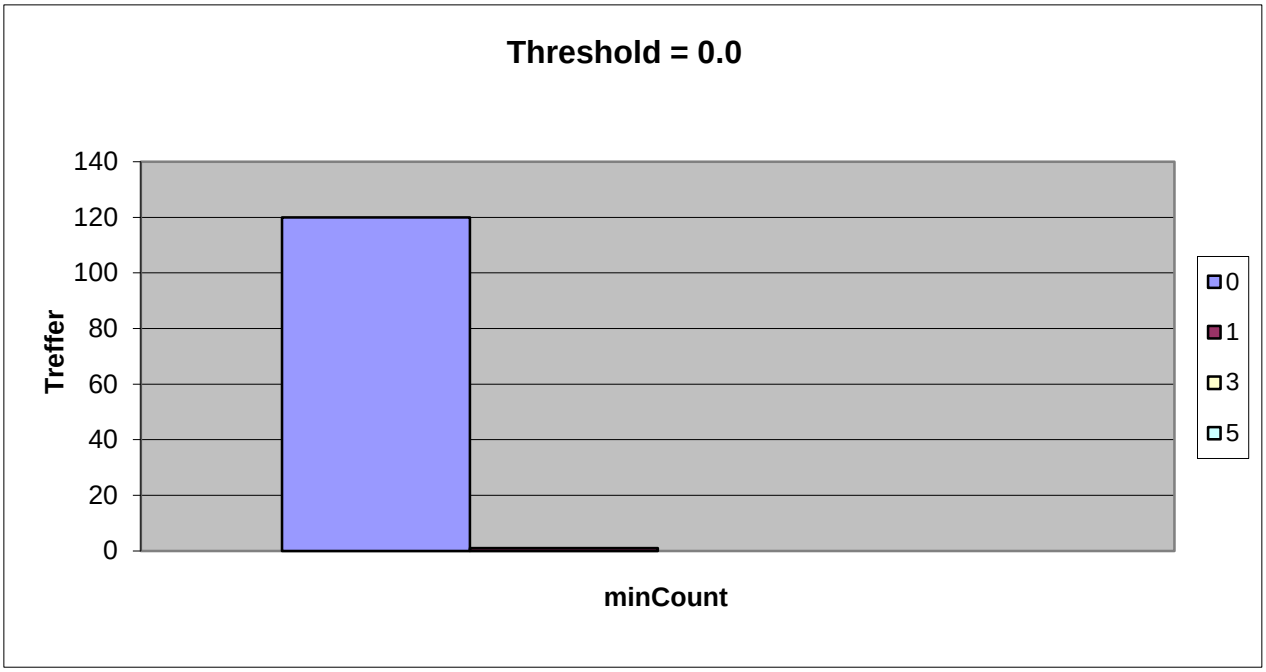
Aus diesen Werten können wir folgende zusammenfassende Übersichten beziehungsweise Diagramme erzeugen:



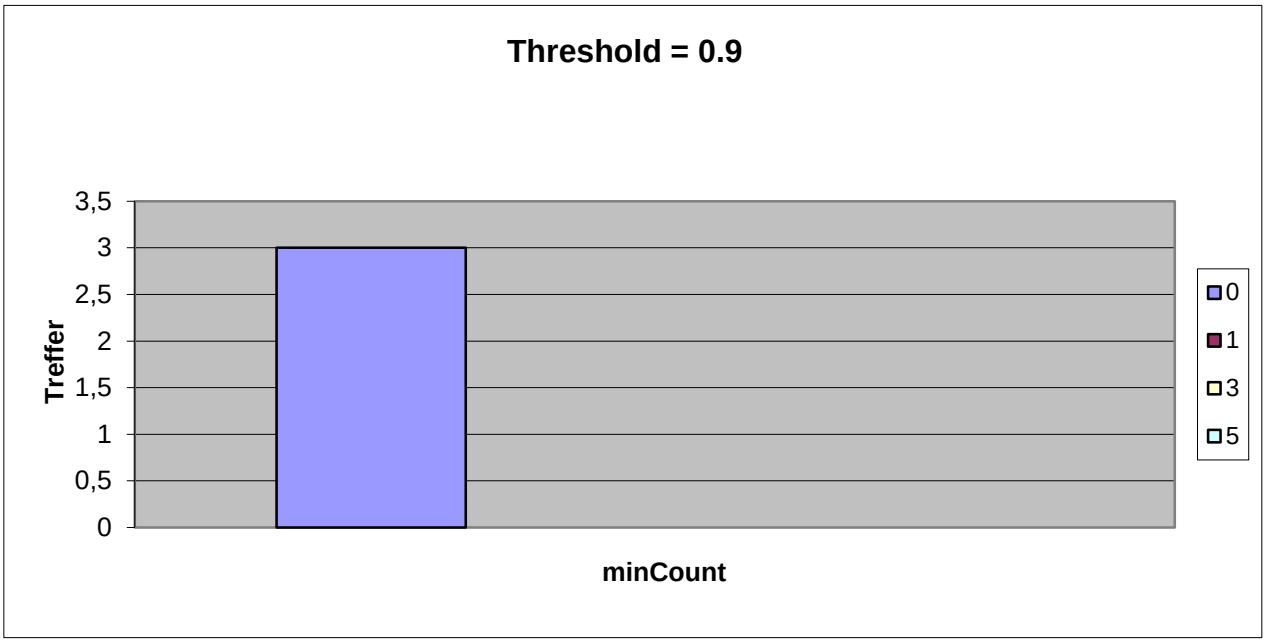
Diese Grafik veranschaulicht die Treffer, die in identischen Dokumenten erzielt werden. Die Einstellung des Thresholds beträgt 0.0. Man erkennt, dass durch einen höher eingestellten Min-Counter die Trefferquote sinkt. Bei einem Wert von 5 werden nur noch 20 Plagiate erkannt.



Dieses Diagramm basiert auf den gleichen Einstellungen. Lediglich der Threshold hat mit 0.9 einen anderen Wert. Es fällt auf, dass sich die Trefferanzahlen bei einer Min-Counter-Einstellung von 0 um mehr als ein Drittel reduziert haben. Bei einer Einstellung von 5 wird das gleiche Ergebnis erzielt.



In dieser Grafik sind die Trefferanzahlen eines Durchlaufes veranschaulicht, bei dem zwei absolut unterschiedliche Dokumente auf Plagiate überprüft wurden. Sobald der Min-Counter einen Wert größer als eins annimmt, reduzieren sich die Treffer auf annähernd null. Die Auflistung der Min-Counter-Werte 3 und 5 entfallen komplett.



Dieses Säulendiagramm bildet das andere Extrem ab, indem der Threshold einen Wert von 0.9 einnimmt. Hier werden lediglich Treffer ausgegeben, sofern der Min-Counter auf 0 eingestellt ist. In allen anderen Fällen (die nicht mehr aufgelistet sind) gibt es keine Treffer und in dem Sinne auch keine Plagiatsvermutungen. Nach wie vor ist es aber schlecht, dass

trotz zweier unterschiedlicher Dokumente, die nicht ansatzweise den gleichen Kontext besitzen, drei mögliche Plagiate entdeckt wurden.

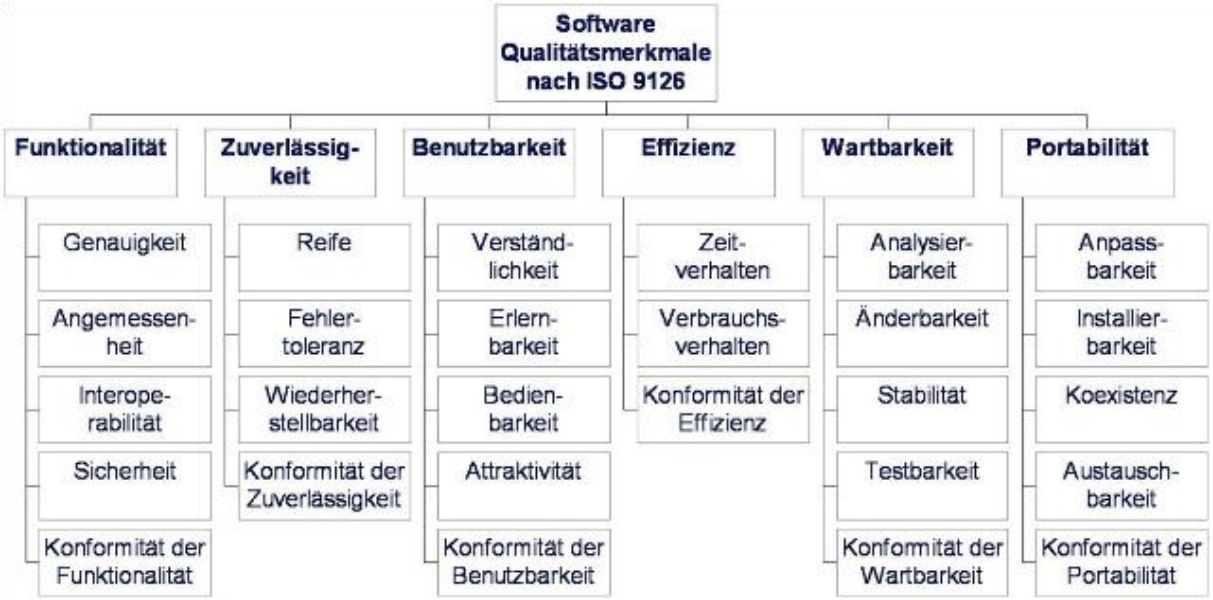
3. Schluss

3.1. Möglichkeiten der Weiterentwicklung

Eine Weiterentwicklung einer Software ist durch die fortschreitende technische Entwicklung unumgänglich. Somit muss jedes Programm derart konzipiert sein, dass es zu jedem Zeitpunkt überarbeitet und ergänzt/ erweitert werden kann. Diesen Anspruch haben wir bei der Softwareentwicklung unseres Projekts von Anfang an berücksichtigt und versucht umzusetzen. Somit war uns eine Übersichtlichkeit und Struktur bezüglich der Implementierung wichtig. Über Schnittstellenkommentare und Klassendokumentationen soll diese Software durch Dritte beziehungsweise den Benutzer möglichst schnell verständlich und nachvollziehbar sein. Somit erhoffen wir uns auch, dass die Voraussetzungen einer Weiterentwicklung unsererseits getroffen wurden.

Neben der Architektur der Software kann auch deren Funktionalität erweitert werden. Die von uns angeführten und benutzen Algorithmen zur Aufdeckung von eventuellen Plagiaten beschränken sich lediglich auf einige Wenige. Somit wäre es durchaus möglich, dass weitere Verfahren zur Plagiatserkennung eingebaut werden können. Ausgehend von dem Naiven Ansatz, dem String-Matching-Algorithmus, lässt sich zum Beispiel der Knuth-Morris-Pratt-Algorithmus erzeugen. Diesem ist wiederrum der Boyer-Moore-Algorithmus sehr ähnlich, wobei hier unter anderem ein anderes Vergleichsverfahren angewandt wird. Insgesamt lässt sich festhalten, dass es, speziell ausgehend vom String-Matching-Algorithmus, viele weitere Varianten gibt, um mögliche Plagiate in Texten zu erkennen. Ebenso lässt sich das Hashwertverfahren in mehreren Hinsichten verändern. So ist es zum Beispiel denkbar, dass man anstelle von Hash-Sets Hashtabellen nutzt, was allerdings immer vom Kontext abhängig ist. Außerdem gibt es weitere Algorithmen, wie zum Beispiel den Rabin-Karp-Algorithmus, der ebenfalls mit Hashfunktionen arbeitet und den man noch implementieren könnte.

Ein weiterer Aspekt, neben den Verfahren zur Plagiatserkennung, den man weiterentwickeln könnte, ist das Laufzeitverhalten. Durch die bereits ausgeführte Lauszeitanalyse wissen wir zwar welcher Algorithmus bei kleineren Datensätzen am schnellsten durchläuft, aber wie sich die Software unter extremer Beanspruchung (sprich großen Datensätzen) verhält, können wir nur schwer einschätzen. Um ein aussagekräftigeres Ergebnis zu erhalten müsste man wohl noch weitere Algorithmen implementieren und alle unter stärkster Belastung arbeiten lassen. Anschließend müsste überprüft werden, inwiefern die schnelleren Verfahren auch eine Effektivität aufweisen, da Schnelligkeit für gewöhnlich noch nichts mit Effektivität zu tun hat.



(Abbildung 5)[8]

Die oben abgebildete Grafik bezüglich der Qualitätsmerkale von Softwaresystemen kann dazu verwendet werden, zu überprüfen, was die eigene Software erfüllt. Aus diesem Grund können wir im Groben die sechs Kriterien durchgehen um Hinweise abzuleiten, in welchen Bereichen unser Softwareprojekt noch zu verbessern beziehungsweise zu erweitern ist.

Unter dem Stichwort Funktionalität ist unter anderem der Bereich der Genauigkeit aufgelistet. Da ein Erkennen eines Plagiates höchste Genauigkeit erfordert und die Einstellungen des Thresholds und des Min-Counters einen sehr großen Einfluss auf die ausgegebenen Ergebnisse hat, ist ein bewusstes Verwenden unserer Software notwendig. In einem solchen Zusammenhang kann an einem Thema wie der Genauigkeit ständig weitergearbeitet werden. Dies wird spätestens bei weiteren Rechtschreibreformen, Verwundung von alter oder inkorrekter Grammatik oder auch Verwendung von weiterer Sprachen unumgänglich.

Dem Bereich der Zuverlässigkeit wird die Kategorie Fehlertoleranz zugeordnet. Eine solche Toleranz ist in unserem Softwareprojekt nicht enthalten. Sobald ein Fehler während des Konvertierungsverfahrens auftritt, wird ein falsches Pattern erzeugt, welches keine korrekten Ergebnisse im Sinne von Plagiaten ausgibt. Weitere Fehler können durch falsch eingestellte Werte des Thresholds oder des Min-Counters entstehen. Dennoch würde die Software durchlaufen und ebenfalls ein unbrauchbares Output erzeugen.

Der Aspekt der Benutzbarkeit, der vor allem die Verständlichkeit, die Erlernbarkeit und die Bedienbarkeit umfasst, sollte bei jeder Softwareentwicklung beachtet werden, da anhand dieser Kategorien ein Projekt über Erfolg oder Misserfolg entschieden wird.

[8] http://www.enzyklopaedie-der-wirtschaftsinformatik.de/Members/herzwurm/QMvS-abb2.png/image_large (Stand: 28.03.2016)

Bezüglich der Effizienz wurde bereits angedeutet, dass die Algorithmen jeweils ein unterschiedliches Zeitverhalten besitzen (siehe Laufzeitanalyse). Deshalb ist davon auszugehen, dass weitere Verfahren existieren beziehungsweise erarbeitet und implementiert werden können, die ein besseres Laufzeitverhalten erreichen.

Die Punkte der Änderbarkeit und der Testbarkeit unter dem Schwerpunkt der Wartbarkeit sind in unserem Falle durchaus gegeben. Gerade das Testen kann mit Hilfe von eigenen ausgedachten Beispielen gut vollzogen werden.

Die Portabilität ist in dem Sinne gegeben, als dass unser Projekt auf sämtlichen, von Java unterstützen, Betriebssystemen läuft. Eine Weiterentwicklung in diesem Bereich ist nicht notwendig.

3.1.1. Webseiten-Analyse

Neben den zuvor genannten Weiterentwicklungsmöglichkeiten der Software, gibt es eine weitere Funktion, die im Zusammenhang mit einer Plagiatserkennungssoftware steht und einen bedeutenden Anteil einnehmen kann. Dadurch, dass ein großer Teil der vergangenen und der heutigen Literatur digitalisiert und per eBook, Newsletter oder allgemein durch das Internet veröffentlicht und bereitgestellt wurde, ist es möglich, den Corpus auf eine unschätzbare Größe zu erweitern. Somit liegt dem Benutzer nicht mehr nur eine Datensammlung auf der eigenen Festplatte zugrunde, sondern er hat die Möglichkeit durch Werken, die ihm selbst unbekannt sind, Plagiate aufzudecken.

Eine mögliche Erweiterung unseres Plagiat-Projekts wäre somit das Miteinbeziehen des Internets in Form einer Webseiten-Analyse. Für diesen Zusatz gibt es mehrere denkbare Herangehensweisen. Eine dafür notwendige Einrichtung ist die Online-Suchmaschine, von denen viele existieren (*Google*, *Bing*, *Yahoo*, *aks.com*, …). Im Folgenden möchte ich kurz auf den Suchalgorithmus an dem Beispiel von Google eingehen, da dieses Verfahren ausschlaggebend dafür sein wird, ob Plagiate gefunden werden können, oder nicht.

Ein Stichpunkt oder ein eingegebener Satz in das Suchfeld der Suchmaschine ist der Beginn eines komplexen Algorithmus, der die Suchergebnisse nicht nur ausgibt, sondern gleichzeitig sortiert. Zwischen den eingegebenen Stichpunkten beziehungsweise den Wörtern des Satzes werden Zusammenhänge festgestellt, nach denen anschließend in einer Datenbank von Google gesucht werden. Diese Datenbank wird laufend von dem Unternehmen aktualisiert und mit weiteren Daten, den Google-Indizes, bestückt. Während dieses Schrittes werden alle relevanten Google-Indizes ausgegeben, zu denen auch abgeänderte Wortstämme und Wortvariationen gehören. Der PageRank-Algorithmus sorgt letztlich dafür, dass die Suchergebisse nach ihrer Relevanz sortiert werden. Dies geschieht dadurch, dass jede Webseite einen Wert zugewiesen bekommt, der sich unter unterem aus der Anzahl der Verlinkungen auf die jeweilige Seite ergibt.

Diesen Suchalgorithmus kann man nun auch dazu verwenden, dass mögliche Plagiate ermittelt werden. Da man zunächst davon ausgehen kann, dass von einem eher wichtigen Werk plagiiert wurde, und im Google-Index sogar studentische Arbeiten enthalten sind, ist die Wahrscheinlichkeit sehr groß, dass eine eingegebene Textpassage durch den Suchalgorithmus von Google einer Webseite zugeordnet werden kann. Diese Webseite gilt es anschließend nur noch zu überprüfen.

Eine Herangehensweise wäre es also nun die Software derart zu erweitern, dass sie im Falle eines nicht gefundenen Plagiats in der eigenen Datensammlung, mit Hilfe einer Online-Suchmaschine nach dortigen Ergebnissen sucht, diese gegebenenfalls auflistet oder eben ausgibt, dass kein Plagiatsverdacht besteht. Hierbei muss allerdings noch beachtet werden, dass die zu überprüfenden Textpassagen nicht zu groß sein dürfen, da die Genauigkeit durch zu viele Wörter abnimmt.

Eine weitere Variation, neben der Möglichkeit eine Suchmaschine zu verwenden, wäre eine tatsächliche Webseiten-Analyse durchzuführen. Eine Voraussetzung ist dabei allerdings, dass eine Webseite gegeben ist, auf der nach Plagiaten zu suchen ist (gegebenenfalls kann diese Voraussetzung durch das vorherig beschriebene Verfahren mit der Online-Suchmaschine erfüllt werden). Die nun vorliegende Internetseite wird in eine Textdatei konvertiert, mit der das Pattern verglichen wird. Hierzu können die bereits enthaltenen Algorithmen, aber auch weitere implementierte Verfahren zur Plagiatserkennung verwendet werden.

3.2. Ergebnisse

Bevor wir auf die Ergebnisse zu sprechen kommen und anschließend mit einer kurzen Reflexion abschließen, möchten wir zunächst noch auf entstandene Probleme und Schwierigkeiten während der Projektarbeit eingehen.

Uns war sehr schnell bewusst, dass ein solches Projekt einen enormen Umfang erreichen kann. Gerade das Entwickeln einer Software kann ein langer Prozess werden, bei dem von vornherein die Grenzen eindeutig gesetzt werden müssen, da man sich sonst in Einzelheiten verliert. Idealerweise handelt es sich um eine zielorientierte Arbeit mit klaren Vorstellungen und einem definierten Rahmen.

In unserem Fall hatten wir das Ziel im Sinne einer Aufgabenstellung erhalten, jedoch war es für uns schwer einzuschätzen, wie umfangreich das Programm werden sollte. Angefangen bei einer grafischen Schnittstelle, über viele bekannte und eher unbekanntere Algorithmen zur Plagiatserkennung, bis hin zu Quelltextkonventionen, hätten wir derart ins Detail gehen können, dass es den Rahmen einer einsemestrigen Projektarbeit gesprengt hätte. Demnach fiel es uns an einigen Stellen schwer uns selbst die Grenzen zu stecken und abzuschätzen, was zu viel beziehungsweise zu wenig wäre.

Eine weitere Schwierigkeit befand sich auf inhaltlicher Ebene. Da wir einen Teil der Software in der Programmiersprache R schreiben sollten, uns aber einig waren, dass wir aus Zeitgründen den übriggebliebenen Teil in Java schreiben, hatten wir zunächst die Herausforderung, Schnittstellen zwischen den Programmiersprachen herzustellen. Eine funktionierende und zusammenhängende Softwareeinheit in unterschiedlichen Programmiersprachen zu schreiben ist zunächst kein Problem, allerdings war uns größtenteils die Sprache R unbekannt, weshalb wir uns zunächst in die Thematik arbeiten und eine Verbindung zwischen dieser Programmiersprache und Java finden mussten. Vor diesem Hintergrund entdeckten wir dann allerdings, dass es in R vorhandene Pakete gibt, die eine Schnittstelle zu anderen Programmiersprachen bilden.

Ein dritter Aspekt war das Ausfindig machen von unterschiedlichen Verfahren zur Plagiatserkennung. Wir hatten schnell Ideen, welche Algorithmen es geben könnte und wie diese zu funktionieren haben, aber abgesehen von dem Hash-Algorithmus, dem naiven „Brute-Force-Ansatz“ und dem Algorithmus, der die Wortliste verwendet, war es zunächst schwieriger als gedacht weitere Verfahren zu finden.

Des Weiteren können die Algorithmen verschieden ausgeführt werden. Über die Möglichkeit blockweise die Algorithmen zu verwenden oder eine komplette Analyse zu betreiben, gab es einige Diskussionen. Dabei sind wir uns einig geworden die Texte komplett zu vergleichen, um möglichst viele Vergleiche erstellen zu können, wodurch kleinere Abweichungen in einem Block nicht zu einer gewaltigen Abweichung führen. Dies ist zwar im Verlauf zeitintensiver, jedoch war unser Ziel möglichst effizient zu arbeiten, was der Hauptgrund für diese Wahl der Implementierung war.

Ebenso die Qualität der Ergebnisse war ein Aspekt, die genauere Betrachtung bedarf. Total-Plagiate stellen kein Problem dar, da bei dem Abgleichen genau dieselben Werte verglichen werden. Das Teilplagiat ist eine größere Herausforderung gewesen.

Zunächst wurden nur einige Sätze verändert und dann das veränderte Dokument mit dem Original verglichen. Dies war kein Problem, sofern die Parameter vernünftig gesetzt wurden (siehe 2.3 Leistungsanalyse). Anschließend haben wir verschiedene Versionen von Wikipedia-Artikeln verglichen. Wenn diese komplett überarbeitet wurden, war es schwierig die Versionen als Plagiate zu betiteln, da der gesamte Kontext neu formuliert worden ist und erweitert wurde. Bei Erweiterungen und Teiländerungen wurde aber auch hier ein Plagiat erkannt. Deutlich wurde, dass verschiedene Autoren eigene Schreibstile verwenden, wodurch ein Text, der die gleiche Thematik behandelt, anders ausfallen kann. Jedoch werden bestimmte Begriffe, die kontextrelevant sind, immer auftauchen und diese müssen dann überprüft werden.

Was uns letztlich weitergeholfen hat, war die Idee, dass das Verfahren zur Spamerkennung in E-Mail-Postfächern im Grunde ähnlich ablaufen müsste. So wird jede ankommende E-Mail unter anderem auf bestimmte Wortstrukturen und Wortzusammenhänge überprüft, was analog dazu während einer Plagiatsüberprüfung passiert. Infolgedessen und anhand

einer Internetrecherche fanden sich schließlich weitere Mechanismen, wie sie auch im vorderen Teil beschrieben sind.

Inhaltlich kann man das Projekt insofern zusammenfassen, als dass die Ausgabe (das Output) von mehreren Faktoren abhängt und beeinflusst wird. So ist entscheidend, welche Werteeinstellung der Threshold und der Min-Counter besitzen und welcher Algorithmus angewendet wurde.

Zusammenfassend verfügt der Hashwert-Algorithmus über ein gutes Laufzeitverhalten, wobei er zudem noch effizient arbeitet. Im Vergleich zu dem String-Matching-Verfahren (der naive Grundalgorithmus), das aus mehreren Gründen als nicht sehr Leistungsstark bewertet werden kann (Laufzeit im Worst Case), sollte das Hashverfahren Vorzug enthalten.

Vom Kontext her betrachtet, hat das Erstellen der zusammenhängenden Software und dieser dazugehörigen Dokumentation viel Zeit in Anspruch genommen. Dadurch, dass wir aus drei verschiedenen Studiengängen kommen, hatten wir auch jeweils eine andere Sicht auf unsere Arbeit, was unserer Meinung dazu geführt hat, dass das Projekt unter diesen Gegebenheiten profitieren konnte. Durch ein gutes Zusammenspiel aus gemeinsamen kooperativen Ideensammlungen und Arbeitsphasen aber auch einzelnen Umsetzungen konnte ein zufriedenstellendes Ergebnis produziert werden.

Die Termine zu Zwischenbesprechungen und zur Zwischenpräsentation waren sehr gut gelegen und kamen uns zeitlich immer sehr passend. So wurden jeweils neue Anreize gesetzt, unsere Fragen beantwortet und die weiteren Schritte diskutiert, die eine gute Orientierung geboten haben. Anschließend hatten wir dann genügend Zeit, um die erhaltenen Informationen und unsere gesetzten Ziele gemeinsam umzusetzen.

Alles in Allem haben wir auf verschiedenen Ebenen viel dazulernen können. Uns Allen ist ein weiteres Mal bewusster geworden, wie genau die Thematik des Plagiarismus genommen werden kann und auch genommen werden sollte und dass es an vielen Stellen eine Verschmelzung der Grenzen zwischen einem Plagiat und einer eigenen geistigen Leistung gibt. All diese Gründe sprechen für dieses Projekt und für die Entwicklung einer entsprechenden Software, die auf dieses Problem aufmerksam macht.

Literaturverzeichnis

Churer Schriften zur Informationswissenschaft (2009): Wissensklau, Unvermögen oder Paradigmenwechsel?. Plagiate als Herausforderung für Lehre, Forschung und Bibliothek. Verlag Arbeitsbereich Informationswissenschaft

Dahmen, W., RWTH Aachen Universität: Mathematisches Praktikum – SoSe 14. Arbeitsblatt

Frakes, W., Information Retrieval (1992): Data structures and algorithms. Verlag Prentice Hall

Krüger, A., Plagiate (2009): Ein ständiges Problem an Universitäten. Studienarbeit. Norderstedt: GRIN Verlag

Internetquellen:

http://www.enzyklopaedie-der-wirtschaftsinformatik.de/Members/herzwurm/QMvS-abb2.png/image_large (Stand, 28.03.2016)

http://plagiat.htw-berlin.de/software/2013-2/ (Stand, 28.12.2015)

https://www.r-project.org/ (Stand, 28.03.2016)

https://wr.informatik.uni-hamburg.de/teaching/wintersemester_2015_2016/parallelrechnerevaluation (Stand, 16.03.2016)