Fakultät Wirtschaftsinformatik und Angewandte Informatik

Otto-Friedrich-Universität Bamberg

Implementierung eines genetischen Algorithmus zur Generierung von Schmerzgrammatiken

Christoph Stocker (Matr. Nr. 1669352)

Projektbericht

WS 2012/13

22. März 2013

Betreuer: Prof. Dr. Ute Schmid, Dipl.-Psych. Michael Siebers

Zusammenfassung

In dieser Arbeit wird die Implementierung eines Klassifikators vorgestellt, der in der Lage ist Sequenzen von Action Units zu erkennen, welche Schmerzausdrücke in der Mimik eines Probanden darstellen. Hierbei wird ein genetischer Algorithmus implementiert. Die Individuen sind dabei Grammatiken, welche die in den Trainingsdaten gegebenen Sequenzen erkennen können. Als Datenstruktur werden hierfür Syntaxbäume regulärer Ausdrücke verwendet. Die resultierende Präzision der Grammatiken bewegt sich einem Bereich von ca. 80 - 100%. Grundlegendes Problem zur Optimierung des Verfahrens ist die Justierung der zahlreichen Parametereinstellungen.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Angesichts einer kontinuierlich alternden Gesellschaft wird die Betreuung pflegebedürftiger Menschen immer wichtiger. Die Entwicklung von computegestützten Systemen, die das Pflegepersonal bei dieser Aufgabe unterstützen, ist daher ein interessantes Forschungsgebiet.
Einer der vielen möglich Teilaspekte dabei ist die Kommunikation mit Menschen, die aufgrund einer Erkrankung nicht oder nur erschwert zu sprachlichen Äußerungen oder Gestik in der Lage sind. Dies kann beispielsweise bei Fällen von fortgeschrittener Demenz der Fall sein.
In vielen Fällen sind reaktive Bewegungen, wie sie zum Beispiel von Schmerzreizen ausgelöst werden, voll funktionsfähig. Teilweise weisen Demenzkranke sogar eine erhöhte mimische Reaktion auf Schmerzreize auf (siehe Lautenbacher et al. (2007)).
Es können daher Systeme entwickelt werden, welche diese Bewegungen erkennen, um die nonverbale Kommunikation mit dem Patienten zu ermöglichen. Die Grundidee besteht dabei darin, Schmerz- oder andere Reize, welche für die Betreuung des Patienten relevant sind, automatisch zu erfassen, auszuwerten und dem Pflegepersonal anzuzeigen.

An der Otto-Friedrich-Universität Bamberg werden für diesen Ansatz Verfahren erforscht. Das Gesicht des Patienten wird dabei mit einer Kamera aufgenommen und die Videodaten auf Muskelbewegungen, welche Schmerz indizieren, untersucht. Hierzu sind zwei Schritte notwendig.
Als erster Schritt müssen Muskelbewegungen aus den Bilddaten erfasst und für weiterführende Verfahren aufbereitet werden. Hierzu gehört die Überführung von Bilddaten in eine maschinell verarbeitbare Repräsentation der Mimik.
In einem zweiten Schritt müssen relevante mimische Ausdrücke von nicht relevanten Ausrücken unterschieden werden. Hierzu können maschinelle Lernverfahren eingesetzt werden, welche beispielsweise aus vorklassifizierten Beispielen erlernen, welche mimischen Ausdrücke Schmerz anzeigen. Im Rahmen dieser Arbeit soll ein solcher Klassifikator für Schmerzausdrücke implementiert werden.

Welche Art von Klassifikatoren für diese Aufgabe verwendet werden kann, hängt stark von der Repräsentation der Mimik ab. Im Rahmen dieser Arbeit wird davon ausgegangen, dass der erste Verarbeitungsschritt bereits durchgeführt wurde. Die hierbei verwendete Darstellung von mimischen Ausdrücken

beruht darauf, verschiedene Muskelbewegungen in Form von sog. *Action Units* zu kodieren. Ein mimischer Ausdruck besteht somit aus einer zeitlichen Abfolge von Action Units. Die Trainingsdaten liegen daher als vorklassifizierte Sequenzen vor, welche Schmerz repräsentieren.
Aufgrund dieser Datenbasis wurde für diese Arbeit entschieden die Mimik als Grammatik darzustellen, welche die Sequenzen der Trainingsdaten erzeugen kann. Zur Generierung der Grammatiken soll hierbei ein genetischer Algorithmus verwendet werden.

Diese Arbeit gliedert sich demnach in vier Teile. In Kapitel 2 werden Ergebnisse aus früheren Arbeiten zur Erkennung von Schmerzmimiken betrachtet und dem in dieser Arbeit dargelegten Vorgehen gegenübergestellt. Zudem wird kurz das Prinzip von genetischen Algorithmen vorgestellt.

Die Umsetzung des Klassifikators sowie dessen Implementierung wird in Kapitel 3 vorgestellt. Besonderer Augenmerk soll hierbei darauf gelegt werden, wie der genetische Algorithmus hinsichtlich der gegebenen Datengrundlage umgesetzt wurde.

Kapitel 4 der Arbeit dient der Evaluation der vorgestellten Umsetzung. Hierbei soll vor allem die Effizienz der Umsetzung in Hinblick auf das Laufzeitverhalten und die Genauigkeit bei der Klassifikation aufgezeigt werden. Des Weiteren sollen auch die Parameter des Algorithmus im Hinblick auf ihre Auswirkungen analysiert werden.

In Kapitel 5 werden die Ergebnisse dieser Arbeit kurz zusammengefasst und ein Ausblick auf mögliche Anknüpfungspunkte an diese Arbeit gegeben. Der Fokus soll hierbei auf mögliche Ansätze zur technischen Erweiterung des vorgestellten Verfahrens gelegt werden.

Kapitel 2

Lernen von Grammatiken aus Sequenzen von Action Units

Die theoretische Grundlage dieser Arbeit bilden die Ergebnisse von Schmid et al. (2012). Das dort beschriebene Verfahren nutzt ABL (siehe van Zaanen (2002)) zur Generierung einer Grammatik, welche Sequenzen von Action Units erkennen kann. Eine Action Unit entspricht dabei einer von 43 möglichen Bewegungen der Gesichtsmuskeln. In Gegensatz zu früheren Ansätzen wird das von Schmid et al. angewendete Verfahren auf Sequenzen von Action Units anstatt auf Mengen angewendet. Der Vorteil ist, dass die zeitliche Abfolge der Action Units erfasst wird, welche eine zusätzliche Informationsquelle über die Art des Reizes darstellt.

Die Kodierung von Schmerzmimiken in Form von Action Units basiert auf den *Facial Action Coding System* (kurz FACS) von Ekman und Friesen Ekman and Friesen (1978). Das von Schmid et al. vorgestellte Verfahren erzielt bereits vielversprechende Resultate. Aufbauend auf dem ABL-basierten Verfahren von Schmid et al. wurde von Javier et al. (2012) in einem studentischen Folgeprojekt eine System zur Grammatikinferenz entwickelt, welches auf Schwarmalgorithmen basiert.
Das dort entwickelte System nutzt das Prinzip der evolutionären Algorithmen, um Grammatiken zu inferieren. Die verwendete Repräsentation der Grammatiken sind Produktionssysteme. Eine Grammatik erkennt eine Sequenz von Action Units genau dann, wenn sie diese erzeugen kann.
Der evolutionäre Ansatz besteht nun darin, mehrere Grammatiken zu generieren, diese zufallsbasiert zu verändern und gemäß einer Fitnessfunktion aufgrund ihrer Effizienz zu bewerten. In weiteren Iterationen dieses Vorgehens werden die Grammatiken als Ausgangsgrundlage verwendet, welche das größte Potential, d.h. die höchste Fitnessbewertung erhalten haben.

Diese Arbeit baut auf die Resultate des Projekts von Javier et al. (2012) auf und erweitert den Schwarm-basierten Ansatz zu einem genetischen Ansatz, welcher die einzelnen Grammatiken als Individuen einer Population betrachtet.

Neben der Mutationsoperation wird eine Austauschoperation hinzugefügt, welche Individuen erlaubt, Teile ihres Erbguts auszutauschen. Unter dem Erbgut eines Individuums versteht man dabei die Grammatik selbst. Wird die Grammatik beispielsweise in Form von Produktionsregeln erstellt, so entspricht ein Crossover einem Austausch von einer oder mehrerer Regeln.
Die Motivation dieser Austauschoperation ist dabei das Crossover, wie es bei der Meiose in den Keimzellen eines biologischen Organismus stattfindet. Ziel des Austauschs ist nützliche Regeln zwischen Individuen zu übertragen und so die Grammatikinferenz effizienter zu gestalten.

Kapitel 3

Implementierung des Klassifikators

3.1 Datenstrukturen

Das Vorgängerprojekt dieser Arbeit verwendet einen regelbasierten Ansatz, um die Grammatiken zu repräsentieren. Der Nachteil dieser Darstellung ist, dass Operationen wie die Mutation, welche die Grammatiken verändern, eine aufwendige Konsistenzprüfung nach sich ziehen, welche den Aufwand einer Veränderung erhöhen (Javier et al., 2012, vgl. Kapitel 4.3).
Teil der Motivation dieser Arbeit ist die Nutzung eines Crossover-Operators, welcher ebenfalls eine Konsistenzprüfung der Grammatik erfordern würde. Aus diesem Grund wurde entschieden, die Grammatiken stattdessen als reguläre Ausdrücke darzustellen.
Ein regulärer Ausdruck ist eine intensionale Beschreibung von Mengen von Zeichenketten. Der Ausdruck selbst ist dabei eine syntaktische Beschreibung, welche die in der Menge enthaltenen Sequenzen erfüllen müssen. Durch reguläre Ausdrücke lassen sich reguläre Mengen beschreiben, d.h. „Mengen, die durch die Operationen der Vereinigung, Konkatenation und Sternbildung entstehen“ (Carstensen et al., 2010, S. 70).
Im Gegensatz zu einer Darstellung durch Produktionsregeln können allerdings nur reguläre Grammatiken dargestellt werden. Dieser Nachteil wurde angesichts der entfallenden Konsistenzprüfung in Kauf genommen.

3.1.1 Implementierung der Grammatiken

Um reguläre Ausdrücke adäquat darstellen zu können, wurde entschieden, diese als Syntaxbäume zu implementieren. Jeder Knoten im Baum entspricht dabei einem Operator eines regulären Ausdrucks. Jede Kante entspricht einem Argument, welche die Operatoren annehmen können.
An sich könnte man reguläre Ausdrücke als simple Zeichenketten darstellen. Die Repräsentation als Syntaxbäume hat jedoch den deutlichen Vorteil, dass diese gut mit genetischen Algorithmen harmonieren. Vorteilhaft ist vor allem, dass die Mutation sowie das Crossover effizient auf Syntaxbäume angewendet werden können.

Die gewählte Darstellung ist stark angelehnt am Prinzip der genetischen Programmierung (Mitchell, 1997, siehe S. 262).
Um die regulären Ausdrücke auswerten zu können, enthält jeder Knoten eine Methode, welche eine Repräsentation des Knotens als herkömmliche Zeichenkette zurückgibt. Durch rekursives Aufrufen dieser Methode auf alle Kindknoten und Konkatenation der Ergebnisse kann der gesamte Syntaxbaum in eine Darstellung als Zeichenkette überführt werden. Diese kann dann durch den in Java bereits verfügbaren Interpreter genutzt werden, um zu ermitteln, ob eine Grammatik eine Sequenz erkennen kann.

3.1.2 Umsetzung der Operatoren

Im Folgenden werden alle Operatoren erläutert, welche im Rahmen dieser Arbeit implementiert wurden.

Blattknoten

Unter Blattknoten werden alle Knoten im Syntaxbaum verstanden, welche keine Kindknoten besitzen können. Diese entsprechen Operatoren welche keine Argumente annehmen. Dazu gehören die Terminalknoten sowie die Leerknoten.
Leerknoten enthalten die leere Zeichenkette und entsprechen somit dem leeren regulären Ausdruck. Sie dienen als Platzhalter für eine spätere Mutation oder Crossover.
Terminalknoten entsprechen einer einzigen Action Unit, z.B. *au3* und bilden somit die Grundlage für alle komplexeren Operatoren.

Verknüpfungsknoten

Für die Verknüpfung mehrerer Operatoren wurden Und-Knoten sowie Oder-Knoten implementiert. Und-Knoten entsprechen dabei der Konkatenation mehrerer Operatoren. Die beiden Terminalknoten *au3* und *au7* können beispielsweise durch einen Und-Knoten zur Sequenz *au3au7* zusammengesetzt werden.

Oder-Knoten entsprechen der Vereinigung mehrerer alternativer Operatoren. Die eben genannten Beispielknoten könnten somit von einem Oder-Knoten zum regulären Ausdruck *au3|au7* zusammengesetzt werden. Damit das Zusammensetzen allerdings richtig durchgeführt werden kann, muss durch Klammerung sichergestellt werden, das nebeneinanderliegende Operatoren die gewünschte Bedeutung nicht verändern.
Ist beispielsweise der eben genannte Oder-Knoten Argument eines Stern-Quantors, so ergibt sich der reguläre Ausdruck *(au3|au7)**. Würde man die Klammerung vernachlässigen, so ergäbe sich der Ausdruck *au3|au7**, welcher nicht äquivalent zum vorherigen Ausdruck ist.
Des Weiteren wurde aus technischen Gründen entschieden nicht-gruppierende Klammerungen (*Non-capturing groups*) zu verwenden. Da Gruppierungsinformationen für die in dieser Arbeit vorgenommene Implementierung nicht relevant sind, kann dadurch der Interpreter entlastet werden.

Quantoren

Es wurden die drei üblichen Quantoren *?*, *+* sowie *** implementiert. Wie bereits bei den Verknüpfungsknoten müssen auch die Argumente der Quantoren geklammert werden, um die gewünschte Bedeutung nicht zu verändern. Eine quantifizierte Action Unit wäre demnach zum Beispiel *(au3)**.
Ein wichtiges Detail bei der Implementierung der Quantoren ist, dass anstatt der herkömmlichen Quantoren die possessive Variante verwendet wird. Der Unterschied zum normalen Quantor besteht darin, dass possessive Quantoren im Laufe des Matching einmal gefundene Teilsequenzen nicht wieder freigeben, d.h. kein Backtracking erlauben.
Dadurch kann es dazu kommen, dass der reguläre Ausdruck Sequenzen nicht erkennt, wenn mehrere konkatenierte Quantoren die selben Sequenzen fordern. Durch Backtracking kann der Operator eine einmal gefundene Teilsequenz wieder freigeben und damit einen Match der gesamten Sequenz ermöglichen.
Allerdings entspricht Backtracking einer Suche mit rekursiven Funktionsaufrufen und steigert somit die Laufzeit exponentiell mit der Größe des Suchbaumes. In vielen Fällen können Sequenzen dann nicht mehr in akzeptabler Laufzeit erkannt werden, wenn komplexe Konstrukte wie sehr breite Oder-Knoten quantifiziert werden.

3.2 Algorithmen

Die Implementierung richtet sich stark nach der von Mitchell beschriebenen Variante des genetischen Algorithmus (Mitchell, 1997, S. 251). Die wichtigsten Schritte des Verfahrens sind demnach die Selektion, die Mutation sowie das Crossover. Zudem ist die Implementierung der Fitnessfunktion ein weiteres Schlüsselelement.

3.2.1 Selektion

Für die Selektion wurde die von Mitchell präsentierte probabilistische Variante verwendet. Die Selektion eines Individuums h_i erfolgt nach einer Gewichtung gemäß der Fitness, bezogen auf die aufsummierte Fitness der gesamten Population:

$$Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{p} Fitness(h_j)}$$

Es ist anzumerken dass es noch weitere Möglichkeiten gibt, die Selektion zu implementieren. Alternativen wären beispielsweise das Tournamentverfahren oder die Selektion nach Rang (Mitchell, 1997, S. 255f.).

3.2.2 Fitness

Die Fitnessfunktion bewertet die Effizienz eines Individuums. Es ist daher naheliegend die Präzision einer Grammatik als Maß der Fitness in Betracht zu ziehen. Ein Individuums ist folglich umso fitter, je mehr Sequenzen der Trainingsdaten es erkennen kann.

Aus der Umsetzung der Grammatiken sowie einigen technischen Aspekten ergeben sich allerdings noch weitere Kriterien welche für die Bewertung einer Grammatik in Betracht gezogen werden können.
Zum einen erhöhen große Grammatiken signifikant die Laufzeit des Algorithmus. Der Grund hierfür ist neben der steigenden Anzahl an Knoten im allgemeinen die steigenden Komplexität, welche sich durch Oder-Knoten sowie Quantoren ergibt. Enthält eine Grammatik beispielsweise viele Oder-Knoten, so müssen beim Matching der regulären Ausdrücke für jede Veroderung alternative Möglichkeiten in Betracht gezogen werden. In diesem Fall steigt die Laufzeit demzufolge exponentiell in der Größenordnung $O(b^d)$, wobei b die durchschnittliche Breite und d die durchschnittliche Tiefe der Syntaxbäume bezeichnet. Dies kann durchaus dazu führen, dass die Laufzeit des Algorithmus nicht mehr im akzeptablen Bereich liegt. Demzufolge ist es sinnvoll, Grammatiken niedriger zu bewerten, welche viele Oder-Knoten enthalten.
Zum anderen lässt sich über die Fitnessfunktion das Generalisierungsverhalten des Algorithmus steuern. Grundsätzlich lässt sich festhalten, dass durch die Verwendung von regulären Ausdrücken als Repräsentationsgrundlage sowie durch das Fehlen negativer Trainingsbeispiele im Prinzip jede Sequenz durch einen einfachen regulären Ausdruck erkennen lässt:

$$(au1|au2|au3|au4|\ldots|auN)^*$$

Dieser Ausdruck liefert zwar für jede Sequenz eine Präzision von 100%, ist aber nicht sehr aussagekräftig, da er Sequenzen, welche Schmerz anzeigen, nicht von anderen Sequenzen unterscheiden kann. Eine Klassifikation ist damit folglich nicht mehr möglich.
Aufgrund des Mangels an negativen Trainingsdaten lässt sich dieses Phänomen nicht vollständig verhindern. Allerdings kann das Generalisierungsverhalten des Algorithmus auf ein Mindestmaß eingeschränkt werden, indem extrem generalisierende Ausdrücke vermieden werden. Der oben genannte Ausdruck lässt sich beispielsweise vermeiden, indem Grammatiken niedrig bewertet werden, welche viele Veroderungen enthalten.
Hierbei ist jedoch anzumerken, dass dies das Voranschreiten der Evolution negativ beeinflusst, da potenziell wertvolle Oder-Knoten bestraft werden, wenn sie noch nicht die passenden Kindknoten enthalten, um die Präzision zu erhöhen. Es sind daher mehr Iterationen notwendig, um die Präzision zu erhöhen, was die Laufzeit erhöht.
Des Weiteren macht es gemäß dem Sparsamkeitsprinzips (*Ockhams Rasiermesser*) durchaus Sinn, die Anzahl der Knoten eines Syntaxbaums ebenfalls in die Bewertung einfließen zu lassen. Von zwei Grammatiken, welche die gleiche Präzision erzielen, wird demzufolge diejenige bevorzugt, welche weniger Knoten enthält und damit tendenziell eher spezifischer ist.
In die Implementierung der Fitnessfunktion wurden alle genannten Aspekte miteinbezogen. Die endgültige Fitness einer Grammatik h besteht somit aus der gewichteten Summe aller Teilaspekte:

$$f(h) = \frac{\sum a_i w_i}{\sum w_i}$$

Hierbei bezeichnet a_i eine Teilaspekt, sowie w_i das dazugehörige Gewicht. Jene können über die jeweiligen Parameter in der Populationsklasse eingestellt werden, um die Berechnung der Fitness an die gewünschten Bedingungen anzupassen.
Die Präzision einer Grammatik wird direkt anhand der Trainingsdaten berechnet und liegt somit in Intervall $[0, 1]$. Der Aspekt der vorhandenen Oder-Knoten kann einfach quantifiziert werden, indem man die Anzahl der Oder-Knoten einer Grammatik der Gesamtanzahl an Knoten gegenüberstellt. Dieser Aspekt liegt demzufolge ebenfalls im Bereich $[0, 1]$. Für den Aspekt der verwendeten Quantoren wird ebenso verfahren.
Der Aspekt der Länge berechnet sich aus $\frac{1}{1+n}$, wobei n die Gesamtanzahl der Knoten einer Grammatik bezeichnet. Demzufolge liegt er im Bereich $[0.5, 0[$.
Erhält jeder Aspekt eine Gewichtung von 1, so liegt die Fitness einer Grammatik im Intervall $[0.5, 2.5]$. Wie genau die Einstellung der Gewichte vorzunehmen ist, hängt von den Trainingsdaten sowie von den Anforderungen an die Laufzeit ab.

3.2.3 Mutation

Eine Mutation wird für jede in der Population vorhandene Grammatik mit einer vorgegebenen Wahrscheinlichkeit angewendet. Dieser Parameter kann in der Populationsklasse eingestellt werden. Da eine Grammatik als ein Syntaxbaum repräsentiert wird, ist die naheliegendste Form der Mutation, einen beliebigen Knoten im Syntaxbaum durch einen neuen zu ersetzen.

Ein Beispiel für dieses Vorgehen ist in Abbildung 3.1 dargestellt. Die Action Unit *au3* wird durch einen Und-Knoten ersetzt. Hier zeigt sich bereits das breite Spektrum an Entscheidungen auf, welche bei der Implementierung der Mutation auftreten.
Wird wie im genannten Beispiel eine Action Unit durch einen Und-Knoten ersetzt, so muss beispielsweise entschieden werden, welche Kindknoten im Und-Knoten eingefügt werden. Eine mögliche Option wäre es beispielsweise, die alte Action Unit sowie einen leeren Knoten einzufügen. Die resultierende Grammatik hätte demnach die selbe Aussagekraft wie die ursprüngliche, hätte jedoch durch den zusätzlichen Leerknoten weiteres Potenzial durch zukünftige Mutation eine höhere Präzision zu erreichen.
Eine andere Option wäre es zufällig generierte Knoten als Kindknoten einzusetzen. Dies bewirkt einen größeren Sprung im Suchraum, erzeugt jedoch mit einer größeren Wahrscheinlichkeit ineffiziente Grammatiken.

Hat man sich für gewisse Varianten der Mutation entschieden, so ist zudem festzulegen, mit welcher Wahrscheinlichkeit welche Variante zu verwenden ist. So hat es sich im Laufe der Arbeiten an diesem Projekt beispielsweise erwiesen, dass Mutationen, welche neue Teilbäume erzeugen, nur sehr selten vorgenommen werden sollten (Größenordnung 1-2%). Werden zu häufig neue Teilbäume erstellt, so entstehen sehr schnell breite Syntaxbäume, was die Laufzeit signifikant erhöht.
Insgesamt sind alle dieser Entscheidungen als Parameter der Implementierung zu betrachten, da sie Wirkungsweise des Algorithmus sowie die Laufzeit be-

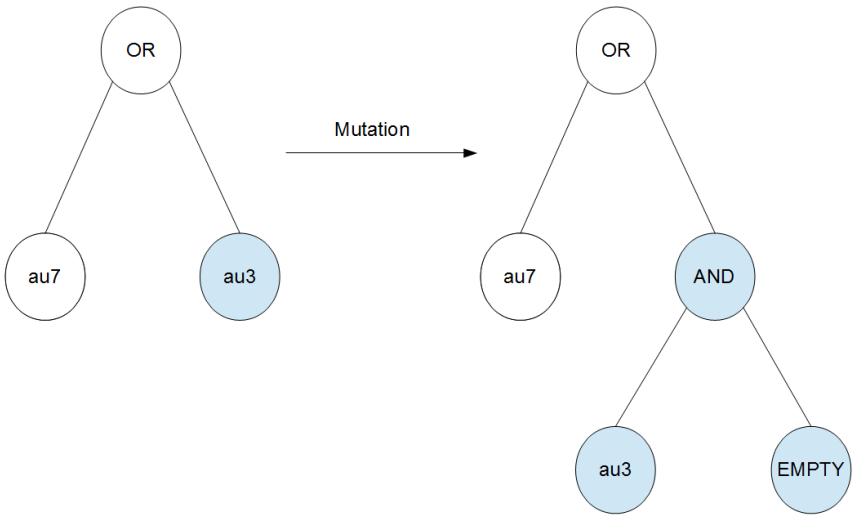

Abbildung 3.1: Beispiel für eine Mutation

einflussen. Zwecks der einfacheren Lesbarkeit wurde jedoch darauf verzichtet, die einzelnen Detailparameter der Mutation bei den in der Populationsklasse befindlichen Parametern des Algorithmus anzugeben. Stattdessen sind die einzelnen Mutationsmöglichkeiten und deren Wahrscheinlichkeiten in der Mutation angegeben. Weitere Details sind daher im dokumentierten Quellcode zu finden.

Neben der Mutation an sich wurden zwei weitere Aspekte implementiert, welche sich empirisch als nützlich erwiesen haben. Zum einen wurde die Möglichkeit implementiert, bei der Mutation neu erstellte Knoten bis zu einem gewissen Grad ebenfalls zu mutieren, so dass eine einzige Mutation einer Grammatik sehr komplexe Änderungen enthalten kann (im Rahmen dieser Arbeit als *Deep Mutation* bezeichnet). Dies entspricht nicht dem Prinzip, welchem die Mutation zugrunde liegt. Orientiert man sich stark am Vorbild der Natur, so sollten Mutationen nur sehr kleine Veränderungen bewirken, um nicht allzu große Sprünge im Suchraum zu bewirken.
Es hat sich jedoch im Laufe der Arbeit erwiesen, dass weitläufige Mutationen bessere Ergebnisse erzielen. Die genaue Ursache konnte im Rahmen dieser Arbeit nicht evaluiert werden, daher wurden entsprechende Parameter in der Populationsklasse implementiert, um die *Deep Mutation* nur bei Bedarf aktivieren zu können.
Zum anderen hat es sich ebenfalls als nützlich erwiesen, die Mutation einer Grammatik mehrfach zu wiederholen, und nur das beste Resultat als Ergebnis in die Population zu übernehmen. Die Laufzeit wird dadurch zwar deutlich erhöht, die dabei erzielte Präzision ist jedoch deutlich höher als bei der herkömmlichen

Methode. Die Anzahl der durchgeführten Mutationen kann ebenfalls über den dazugehörigen Parameter der Populationsklasse bestimmt werden.

3.2.4 Crossover

Im Gegensatz zur Mutation gestaltet sich das Crossover bei der genutzten Repräsentation als Syntaxbäume relativ einfach. Ein Crossover wird durchgeführt, indem von zwei Grammatiken jeweils zufallsbasiert ein Teilbaum ausgewählt wird, welche dann ausgetauscht werden. Ein Beispiel für das Crossover bei Syntaxbäumen ist bei Mitchell (Mitchell, 1997, S. 263) zu finden.
Die Anzahl an Grammatiken, welche durch Crossover in die nächste Population übernommen werden sollen wird durch den dazugehörigen Parameter in der Populationsklasse gesteuert. Welche Grammatiken letztendlich durch Crossover verändert werden sollen wird zufallsbasiert, jedoch gewichtet nach der jeweiligen Fitness entschieden. Die dazu verwendete Berechnungsvorschrift entspricht der Formel aus Kapitel 3.2.1.

Kapitel 4

Parametereinstellungen und Evaluation

Aufgrund der vorhandenen Datenbasis zieht der Algorithmus nur positive Trainingsbeispiele in Betracht. Dies hat zur Folge, dass nur die Vollständigkeit, nicht jedoch dich Korrektheit der Implementierung evaluiert werden kann. Zur Evaluation der Vollständigkeit soll zunächst die Präzision betrachtet werden.
Des Weiteren hat die hier vorgestellte Implementierung eine Reihe von Parametern aufzuweisen, die sich sowohl auf das Resultat als auch auf die Laufzeit des Algorithmus auswirken. Die Auswirkungen dieser Parametereinstellungen werden daher ebenfalls genauer erläutert.

4.1 Präzision

Die Vollständigkeit des Algorithmus wird anhand der Präzision gemessen. Unter Präzision ist hierbei der prozentuale Anteil an Trainingsbeispielen zu verstehen, der von einer Grammatik erkannt werden kann. Ausschlaggebend für die Präzision im Ganzen ist infolgedessen die höchste erreichte Präzision in der Population bei Abbruch des Algorithmus.
Im Rahmen dieser Arbeit wurde mit zwei Datensätzen gearbeitet, die unterschiedliche Schwierigkeitsgrade aufgrund ihrer Komplexität aufweisen. Die Eckdaten der beiden Datensätze sind in Tabelle A.1 aufgelistet. Der Hauptunterschied der Datensätze besteht in der Anzahl der Sequenzen, sowie in der Anzahl der Action Units, die verwendet werden. Datensatz 1 ist aufgrund dieser Einteilung als der „einfache“ Datensatz zu verstehen, sowie Datensatz 2 als der „schwierige“ Datensatz.

Es ist anzumerken, dass dies nur eine grobe Unterteilung darstellt, die nicht effektiv auf die Komplexität der Daten hinweisen kann. Da es sich um eine rein statistische Einteilung handelt wird darin beispielsweise nicht berücksichtigt, ob die enthaltenen Sequenzen nur mit kontextsensitiven Regeln zu erkennen sind. Einteilungen nach Komplexität dieser Art müssten für genauere Untersuchungen berücksichtigt werden.

Um die Vollständigkeit des Algorithmus auszuwerten wurden nun auf jeden Datensatz mehrere Iterationen ausgeführt. Da es sich um ein randomisiertes Verfahren handelt, liefert die mehrmalige Wiederholung des Algorithmus aussagekräftigere Ergebnisse als eine einfache Durchführung, da jede Iteration leicht unterschiedliche Ergebnisse erzielen kann. Bei mehreren Iterationen kann dann über statistische Werte wie etwa die Standardabweichung der erreichten Präzision Rückschlüsse auf die Stabilität des Algorithmus getroffen werden.
Für die Evaluation wurden mit jedem Datensatz 10 Iterationen durchgeführt. Die Parametereinstellungen dieser Durchläufe sind in Tabelle A.2 aufgeführt. Eine Übersicht über die Ergebnisse der Testläufe ist in den Tabellen A.4 und A.5 zu finden. Im Allgemeinen lässt sich feststellen, das die hier vorgestellte Implementierung die beiden Datensätze recht gut erkennen kann.

Für Datensatz 1 wurde der Zielwert von 95% Präzision bei jedem Durchgang erreicht. Hierbei waren, bis auf eine Ausnahme, nie mehr als 100 Iterationen notwendig. Die Gesamtdauer für eine Versuchsreihe bewegt sich im Bereich von ca. 25 Sekunden bis hin zu ca. 5 Minuten und ist demzufolge äußerst akzeptabel.

Die Ergebnisse für Datensatz 2 sind ebenfalls als erfolgreich zu erachten. Die Präzision erreicht den erstrebten Zielwert von 95% in 5 der 10 durchgeführten Testläufe. Die restlichen Resultate bewegen sich in einem Bereich von ca. 78% bis 90%.
Um diese Ergebnisse zu erreichen, waren bei Datensatz 2 jedoch deutlich mehr Iterationen erforderlich. In 5 von 10 Fällen erreichte der Algorithmus das festgelegte Maximum von 200 Iterationen. Logischerweise handelt es sich bei diesen 5 Testläufen um diejenigen, die nicht den gewünschten Grenzwert erreichten. Für die erfolgreichen Durchläufe waren jedoch ebenfalls stets deutlich mehr als 100 Iterationen notwendig. Die dafür notwendige Laufzeit ist ebenfalls deutlich höher als bei Datensatz 1. Die Gesamtlaufzeit bewegt sich im Bereich von ca. 10 Minuten bis hin zu über einer halben Stunde.
Ob dies als akzeptabel zu betrachtet ist, hängt vom Einsatzgebiet der Implementierung ab. Für Anwendungen die nicht in Echtzeit ablaufen sollte die notwendige Zeit jedoch im Rahmen liegen.

Um das generelle Verhalten der Implementierung bezüglich Laufzeit und Präzision evaluieren zu können, wurde in jeder Testreihe die pro Iteration benötigte Zeit sowie die dabei erreichte Präzision festgehalten. Die Resultate sind im Appendix angegeben. Abbildung A.1 und A.5 stellen die durchschnittlich erreichte Präzision pro Iteration dar. Die Standardabweichung ist dabei als Y-Balken mit angegeben.
Bei Betrachtung dieser beiden Abbildungen ist zu erkennen, dass unabhängig vom Schwierigkeitsgrad der Daten ca. die erste Hälfte der Iterationen eine stetige Verbesserung der Präzision erzielt, während die zweite Hälfte zur „Perfektion“ der gefundenen Grammatik dient. Betrachtet man die Standardabweichungen bzw. die Einzelverläufe in den Abbildungen A.2 und A.6, so stellt man fest, dass die Testreihen nicht unbedingt gleichmäßig verlaufen. Dies lässt sich durch lokale Maxima erklären, welche erst im Zuge der Evolution überwunden werden müssen. Da der Algorithmus durch Mutation und Crossover stark randomisiert, treten diese lokalen Maxima nicht bei jedem Durchlauf auf.

In Abbildung A.3 und A.7 ist die mittlere Dauer jeder Iteration, sowie die dazugehörige Standardabweichung angegeben. Hier lassen sich wiederum zwei Phasen im Ablauf des Algorithmus erkennen. In der ersten Phase, welche ca. das erste Viertel der Iterationen umfasst, steigt die benötigte Zeit pro Iteration kontinuierlich und ohne große Abweichung an. Dies lässt sich durch die geringe Größe der Syntaxbäume zu Anfang des Algorithmus erklären. Kleine Bäume enthalten weniger Oder-Knoten und Quantor-Knoten als große Bäume, und können dementsprechend schnell ausgewertet werden. Daher ist keine große Abweichung in den einzelnen Testreihen zu erkennen.
In der zweiten Phase steigt die durchschnittliche Dauer weiterhin an, weist jedoch zusätzlich eine größere Streuung unter den Testreihen auf. Grund hierfür ist die steigende Anzahl an Oder-Knoten und Quantor-Knoten. Sind diese ungünstig ineinander verschachtelt, entstehen komplexe Suchbäume, welche vom Interpreter durchsucht werden müssen (siehe 3.1.2). Besonders deutlich wird dies bei Betrachtung von Abbildung A.8. Die Testreihen R3, R6 sowie R10 weisen deutliche lokale Anstiege in der benötigten Zeit auf. Dies entspricht genau dem erklärten Phänomen. Würde die Komplexität der Daten diese hohe Laufzeit erfordern, dürfte es keine Datenreihen geben, welche nicht diese lokalen Spitzen aufweisen.

4.2 Vorhandene Parameter

Generell hat der Algorithmus ein sehr effizientes Laufzeitverhalten. Selbst Grammatiken mit mehreren hundert Knoten können noch in wenigen Millisekunden ausgewertet werden. Jedoch gibt es einige Einstellungen welche sich spürbar auf die Laufzeit auswirken. Die Auswirkungen der einzelnen Parameter sollen im Folgenden genauer erläutert werden.

4.2.1 Anzahl an Individuen

Eine höhere Anzahl an Individuen erhöht die Chance, schneller eine bessere Grammatik zu finden, da jedes Individuum einem Punkt im Suchraum entspricht. Weitere Punkte im Suchraum bedeuten somit einen größeren Teil an überprüften möglichen Hypothesen. Allerdings erkauft man sich diesen Vorteil mit einer Erhöhung der Laufzeit.

4.2.2 Mutationsrate

Die Mutationsrate beschreibt die Wahrscheinlichkeit, mit der eine Grammatik im Zuge einer Iteration mutiert wird. Die Mutation selbst erfolgt in linearer Laufzeit, allerdings müssen nach der Mutation die Präzision und Fitness der Grammatik neu berechnet werden. Eine erhöhte Mutationsrate hat demnach auch eine Erhöhung der Laufzeit zur Folge.
In der Literatur zu genetischen Algorithmen wird oftmals eine niedrige Mutationsrate empfohlen. Dies macht schon allein aus evolutionärer Sicht Sinn, da in der Natur Mutation ebenfalls selten vorkommt. Allerdings hat sich im Rahmen der Arbeit gezeigt, dass eine hohe Mutationsrate zumeist bessere Ergebnisse liefert. Eine Hypothese für dieses Phänomen wäre beispielsweise das starke Generalisierungsverhalten mancher Knoten. So enthalten Grammatiken

in der Regel viele Leerknoten, die in Oder-Knoten enthalten sind. Wird einer dieser Knoten mutiert, so kann die resultierende Grammatik nur eine höhere Präzision aufweisen als die ursprüngliche Grammatik.

4.2.3 Anzahl an Mutationen

Dieser Parameter legt fest, wie oft eine Mutation auf eine Grammatik durchgeführt werden soll. Nur die beste erzielte Mutation wird dabei übernommen. Da jede Mutation eine Neuberechnung der Grammatik zufolge hat, erhöht dieser Parameter bei sehr häufigen Mutationsversuchen deutlich die Laufzeit des Algorithmus.
Allerdings hat sich ebenfalls erwiesen, dass auf diese Weise deutlich schneller eine Verbesserung der Präzision erzielt werden kann. Dies kann ebenfalls mit der erwähnten Hypothese erklärt werden. Häufige Mutationen erhöhen die Wahrscheinlichkeit, einen Leerknoten innerhalb eines Oder-Knoten zu mutieren, was die Präzision nur verbessern kann.

4.2.4 Crossoverrate

Crossover hat analog zur Mutation eine Neuberechnung der entsprechenden Grammatiken zur Folge. Demnach bedeutet eine höhere Crossoverrate auch eine höhere Laufzeit. Weitere Phänomene konnten nicht beobachtet werden. Im Rahmen dieser Arbeit wurde eine eher niedrige Crossoverrate von 20% verwendet.

4.2.5 Gewichte der Fitnessfunktion

Die unterschiedlichen Aspekte der Fitness einer Grammatik lassen sich über die Änderung der jeweiligen Gewichte justieren. Genauere Details zur Fitness-Funktion sind in Kapitel 3.2.2 zu finden. Die genauen Auswirkungen der Gewichte auf die Präzision sind allerdings sehr komplex und bedürfen einer genaueren Evaluation.
Grundsätzlich lässt sich festhalten, dass die Vermeidung von Oder-Knoten oder Quantor-Knoten die Auswertungen der regulären Ausdrücke einfacher macht, was die Laufzeit pro Iteration verringert. Allerdings hemmt eine zu hohe Vermeidung dieser Knoten ebenfalls die erreichbare Präzision, da diese Knoten notwendig sind, um eine Generalisierung zu erreichen. Dies erhöht wiederum die allgemeine Laufzeit des gesamten Algorithmus, da mehr Iterationen notwendig sind, um die gewünschte Präzision zu erreichen.

4.2.6 Refactoring

Unter Refactoring wird im Rahmen dieser Arbeit das Pruning der in den Grammatiken enthaltenen Syntaxbäume verstanden. Ziel des Prunings ist die Reduzierung der enthaltenen Knoten und damit der Komplexität der Grammatik. Hierbei ist jedoch darauf zu achten, dass die Aussagekraft der Grammatik nicht reduziert wird. Die Präzision einer Grammatik darf demnach nicht durch Pruning reduziert werden. Es wurden zwei verschiedene Arten des Pruning implementiert, welche separat von einander durchgeführt werden können.

Bei der ersten Art des Prunings werden aufeinanderfolgende Quantoren durch einen einzigen Quantor ersetzt. Dies kann durch Beachtung legaler Kompositionen von Quantoren geschehen. Die für diese Arbeit verwendete Kompositionstabelle ist im Appendix in Tabelle A.3 zu finden. Des Weiteren können innerhalb von Oder-Knoten alle Kindknoten entfernt werden, welche den gleichen regulären Ausdruck ergeben. Diese Art des Pruning sollte stets durchgeführt werden, da die Entfernung redundanter Knoten mehr Platz für Knoten schafft, welche die Präzision erhöhen.

Die zweite Art des Pruning entfernt alle Leerknoten einer Grammatik, da diese keine Auswirkung auf die Präzision der Grammatik haben. Da eine Grammatik aufgrund der Implementierung im Normalfall viele Leerknoten enthält, führt diese Art des Refactorings stets zu einer starken Reduzierung der enthaltenen Knoten. Allerdings ist anzumerken, dass Leerknoten als Platzhalter für mögliche Mutationen dienen und daher potentiell wertvoll für die Erhöhung der Präzision sind.
Wird diese Art des Refactorings beispielsweise in jeder Iteration durchgeführt, so ist kaum eine Verbesserung der Präzision möglich, da Leerknoten nicht auftreten können, und somit im Zuge einer Mutation stets ein bereits verwendeter „echter“ Knoten entfernt werden muss. Empirisch hat sich erwiesen, dass diese Art des Refactorings nur durchgeführt werden sollte, sobald die Größe der Grammatiken sich spürbar auf die Laufzeit niederschlägt.

4.2.7 Versteckte Parameter innerhalb der Mutation

Wie in Kapitel 3.2.3 bereits beschrieben hat die Mutation massive Auswirkung auf die Effizienz des Algorithmus. Da innerhalb der Mutation zufallsbasiert entschieden wird, auf welche Art und Weise die Knoten verändert werden, haben die Gewichtungen dieser Entscheidungen starke Auswirkung auf die Laufzeit und Effizienz des Algorithmus.
Erhält beispielsweise die Generierung neuer Kindknoten innerhalb der Oder-Knoten eine hohe Wahrscheinlichkeit, so wird die durchschnittliche Breite der Syntaxbäume deutlich erhöht. Dies erhöht zwar die Generalisierung und somit die Präzision, wirkt sich jedoch auch negativ auf die Laufzeit aus, da zusätzliche Veroderungen bei der Auswertung der regulären Ausdrücke mehr zu beachtende alternative Auswertungen nach sich ziehen.
Anhand dieses Beispiels lässt sich erkennen, dass die Wahrscheinlichkeitsverteilungen innerhalb der Mutation ebenso als Parameter zu betrachten sind. Aufgrund der schieren Menge an möglichen Wahrscheinlichkeitsverteilungen wurden diese Parameter im Rahmen dieser Arbeit nicht genauer untersucht. Die verwendeten Einstellungen für die durchgeführten Durchläufe sind im Quellcode dokumentiert.

4.2.8 Sonstige Parameter

In der Implementierung dieser Arbeit sind einige weitere Parameter zu finden, die im Rahmen dieses Berichtes nicht genauer erwähnt wurden. Bei diesen Parametern handelt es sich um technische Details oder Einstellungen, bei denen keine markante Veränderung des Algorithmus beobachtet werden konnte. Wei-

tere Details zu den hier nicht aufgeführten Parametern sind der Dokumentation des Quellcodes zu entnehmen.

Kapitel 5

Zusammenfassung und Ausblick

In dieser Arbeit wurde die Implementierung eines Klassifikators vorgestellt, der in der Lage ist Sequenzen von Action Units zu erkennen, welche Schmerzausdrücke in der Mimik eines Probanden darstellen. Hierbei wurde ein genetischer Algorithmus implementiert.

In Kapitel 2 wurden die theoretischen Grundlagen aus vorhergehenden Arbeiten erläutert. Im Bezug zum Vorgängerprojekt wurde festgestellt, dass diese Arbeit neben der Mutationsoperation zusätzlich das Crossover verwendet.

Kapitel 3 enthält eine Erläuterung der verwendeten Algorithmen und Datenstrukturen. Im Gegensatz zum Vorgängerprojekt wurden die Grammatiken als Syntaxbäume regulärer Ausdrücke repräsentiert. Der Vorteil dieser Datenstruktur liegt in der Ersparnis der Konsistenzprüfung. Die Implementierung des Algorithmus richtet sich nach dem klassischen genetischen Algorithmus, wie er von Mitchell (1997) vorgestellt wird.

In Kapitel 4 wurde die Implementierung auf ihre Effizienz hin evaluiert. Hierzu wurde die Implementierung auf einen einfachen sowie einen komplexen Datensatz angewendet. Für den einfachen Datensatz konnte in jedem Durchlauf die als Ziel gesetzte Präzision von 95% erreicht werden. Der komplexe Datensatz erfordert eine weitaus längere Laufzeit, konnte jedoch mit einer Präzision von mindestens 78% erkannt werden. In 5 von 10 Fällen konnte ebenfalls das gesetzte Ziel von 95% erzielt werden.
Des Weiteren wurden die Schwierigkeiten der Parametereinstellungen erläutert, da diese die Grundlage für die erzielbare Effizienz sowie die benötigte Laufzeit bilden.

Grundsätzlich konnte der Klassifikator komplett umgesetzt werden und ist somit voll funktionsfähig. Allerdings gibt es durchaus Möglichkeiten, die in dieser Arbeit vorgestellte Implementierung noch zu verbessern.
Beispielsweise wurden nicht alle Operatoren implementiert, welche in regulären Ausdrücken verwendet werden können. Moderne Interpreter nutzen beispielswei-

se Rückwärtsreferenzen. Rückwärtsreferenzen lassen Konstruktionen zu, welche mehr als nur reguläre Sprachen erzeugen können. Eine Implementierung dieses Operators würde demnach die Ausdrucksmächtigkeit der erzeugten Grammatiken erhöhen.
Zudem enthält die vorgestellte Implementierung eine sehr große Anzahl an Parametern. Die idealen Einstellungen für diese Parameter konnten in Rahmen dieser Arbeit nicht ergründet werden. Ein besonderes Augenmerk ist hierbei auf die Mutation zu richten. Diese enthält eine große Anzahl an veränderbaren Einstellungen. Eine Optimierung dieser Einstellungen ist allerdings sehr komplex, da die Auswirkungen einzelner Änderungen nur schwer vorherbestimmt werden können. Die genauere Evaluation der Parameter wäre daher ebenfalls von Vorteil.

Literaturverzeichnis

Carstensen, K.-U., Ebert, C., Jekat, S., Ebert, C., Langer, H., and Klabunde, R. (2010). *Computerlinguistik und Sprachtechnologie: Eine Einfuehrung.* Springer, 3. auflage edition. ISBN 3827420237.

Ekman, P. and Friesen, W. (1978). *Facial Action Coding System: A Technique for the Measurement of Facial Movement.* Consulting Psychologists Press, Palo Alto.

Javier, C. F., Fernando, M., and Martina, M. (2012). Learning grammars with swarm genetic approach. Master Project Cognitive Systems, SS 2012.

Lautenbacher, S., Kunz, M., Mylius, V., Scharmann, S., Hemmeter, U., and Schepelmann, K. (2007). Mehrdimensionale schmerzmessung bei demenzpatienten. *Der Schmerz*, 21:529–538.

Mitchell, T. (1997). *Machine Learning.* McGraw-Hill, New York.

Schmid, U., Siebers, M., Seuss, D., Kunz, M., and Lautenbacher, S. (2012). Applying grammar inference to identify generalized patterns of facial expressions of pain. *Proceedings of the 11th International Conference on Grammatical Inference*, 21:1–6.

van Zaanen, M. (2002). Bootstrapping structure into language: Alignment-based learning. *CoRR*, cs.LG/0205025.

Anhang A

Appendix

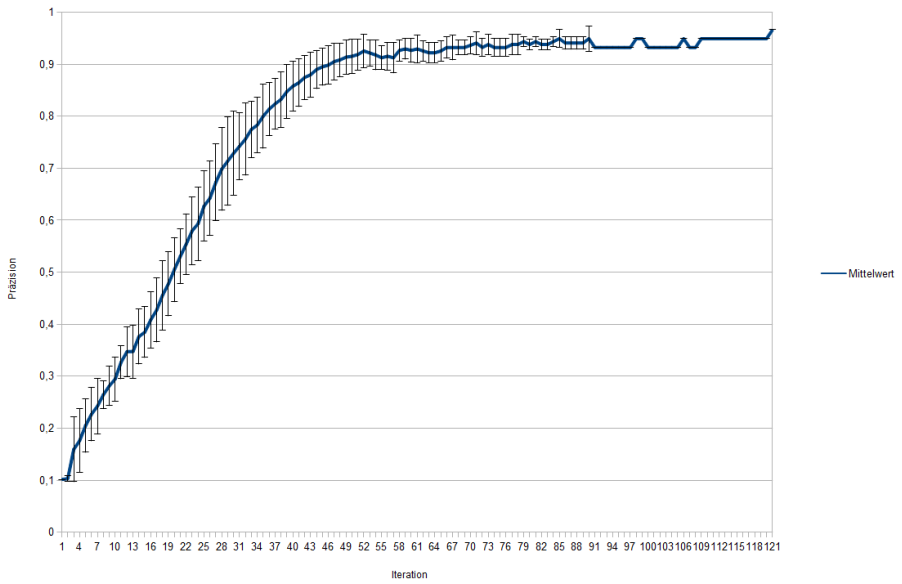

Abbildung A.1: Mittlere Präzision, Datensatz 1

Name	Anzahl AUs	Anzahl Sequenzen	Längste Sequenz	Kürzeste Sequenz	mittlere Sequenzlänge
Datensatz 1	30	59	17	1	3.59
Datensatz 2	76	347	17	1	4.03

Tabelle A.1: Eckdaten der Datensätze

Anzahl Individuen	100
Präzision-Grenzwert	95%
Maximum an Iterationen	200
Crossover-Anteil	20%
Mutationsrate	80%
Mutationsversuche	100
Gewichtung Präzision	1
Gewichtung Quantoren	0
Gewichtung Oder-Knoten	0
Gewichtung Länge	0
Refactoring	Ja
Refactoring leere Knoten	jede 100. Iteration

Tabelle A.2: Parametereinstellungen der Evaluationstestläufe

Erster Quantor	Zweiter Quantor	Komposition
*	*	*
*	?	*
*	+	*
+	+	+
+	*	*
+	?	*
?	*	*
?	+	*
?	?	?

Tabelle A.3: Kompositionstabelle des Refactorings

Versuchsreihe	Iterationen	Dauer [ms]	Mittlere Iterationsdauer [ms]	Präzision
R1	55	29053	538	96.61%
R2	86	73273	862	96.61%
R3	53	28375	545	96.61%
R4	62	29286	480	96.61%
R5	91	76154	846	96.61%
R6	122	279147	2307	96.61%
R7	54	44939	847	96.61%
R8	72	67191	946	96.61%
R9	68	30877	460	96.61%
R10	53	24740	475	96.61%

Tabelle A.4: Ergebnisse der Versuchsreihen, Datensatz 1

Versuchsreihe	Iterationen	Dauer [ms]	Mittlere Iterationsdauer [ms]	Präzision
R1	147	887130	6114	95.10%
R2	200	845073	4246	89.63%
R3	161	1760025	11000	95.39%
R4	200	846135	4251	78.39%
R5	181	1128998	6272	95.10%
R6	165	1467461	8947	95.10%
R7	182	1209238	6680	95.10%
R8	200	1400942	7039	83.86%
R9	200	1481516	7444	84.44%
R10	200	2951795	14833	90.49%

Tabelle A.5: Ergebnisse der Versuchsreihen, Datensatz 2

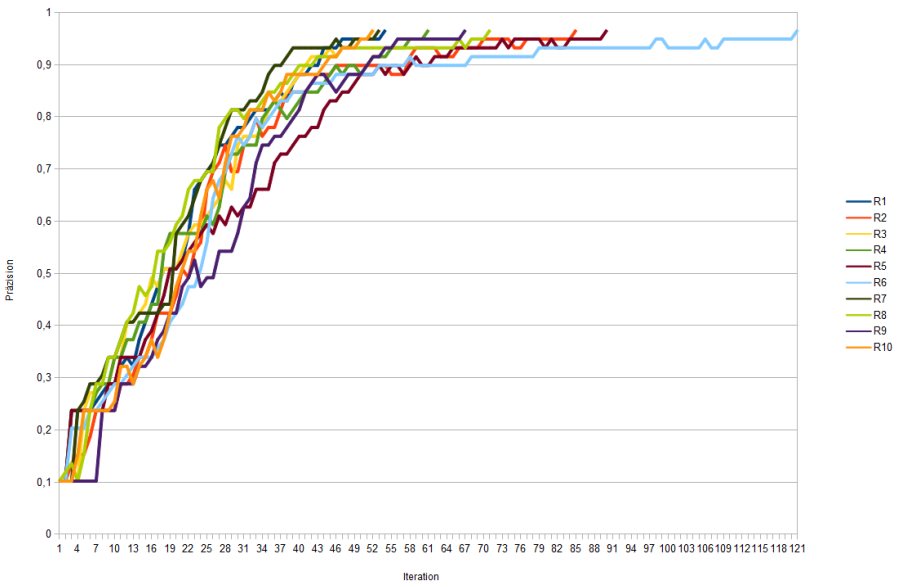

Abbildung A.2: Präzision, Datensatz 1

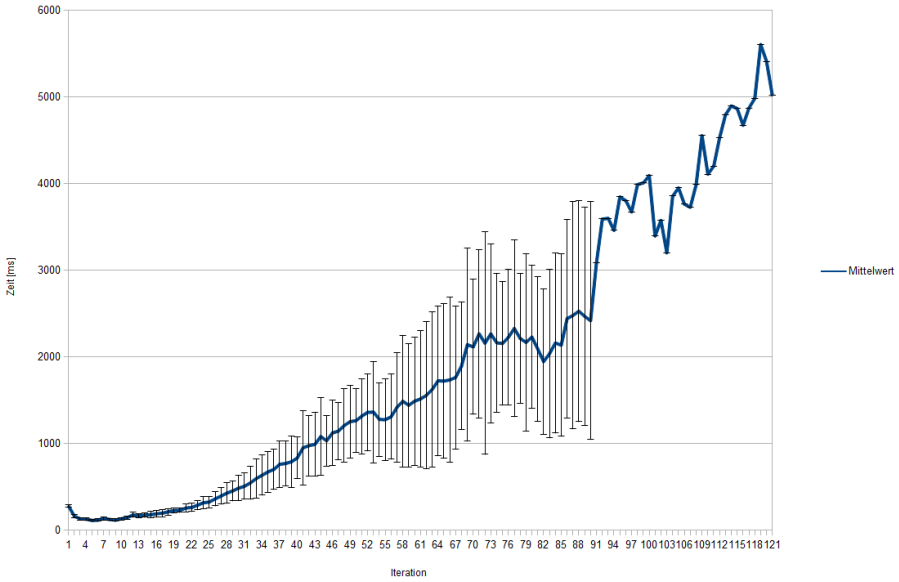

Abbildung A.3: Mittlere Dauer, Datensatz 1

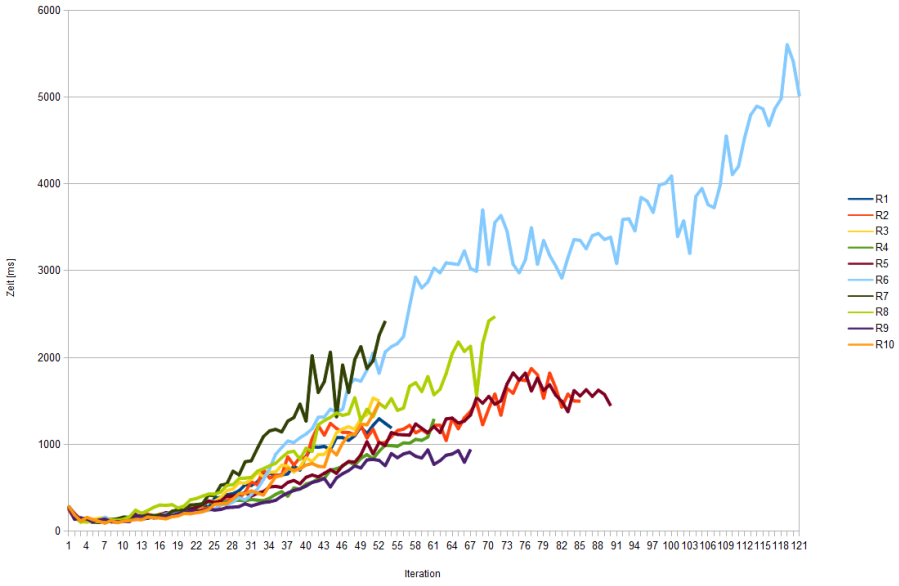

Abbildung A.4: Dauer, Datensatz 1

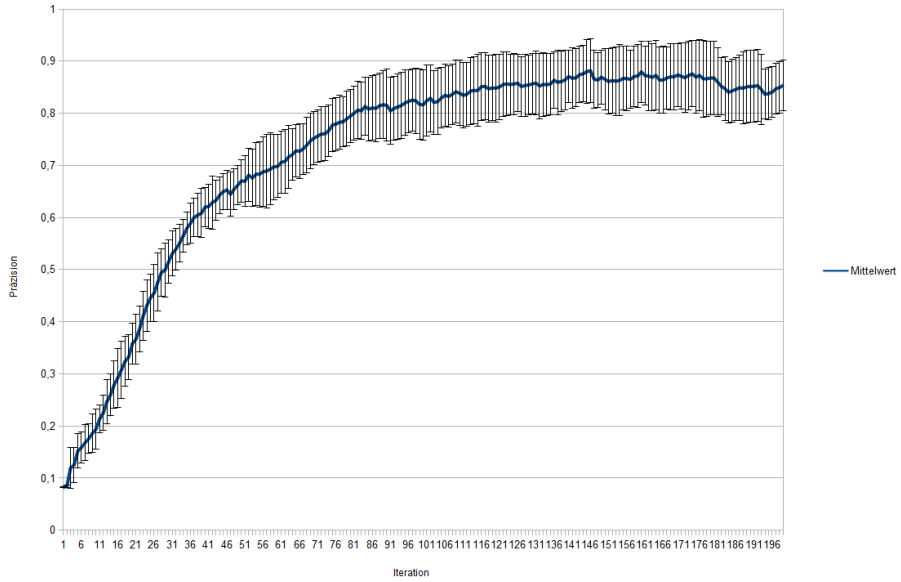

Abbildung A.5: Mittlere Präzision, Datensatz 2

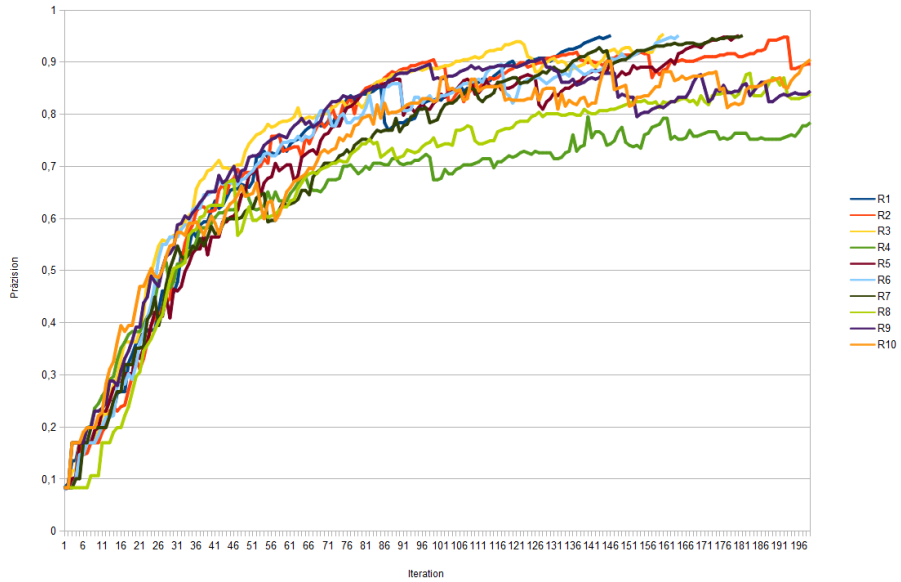

Abbildung A.6: Präzision, Datensatz 2

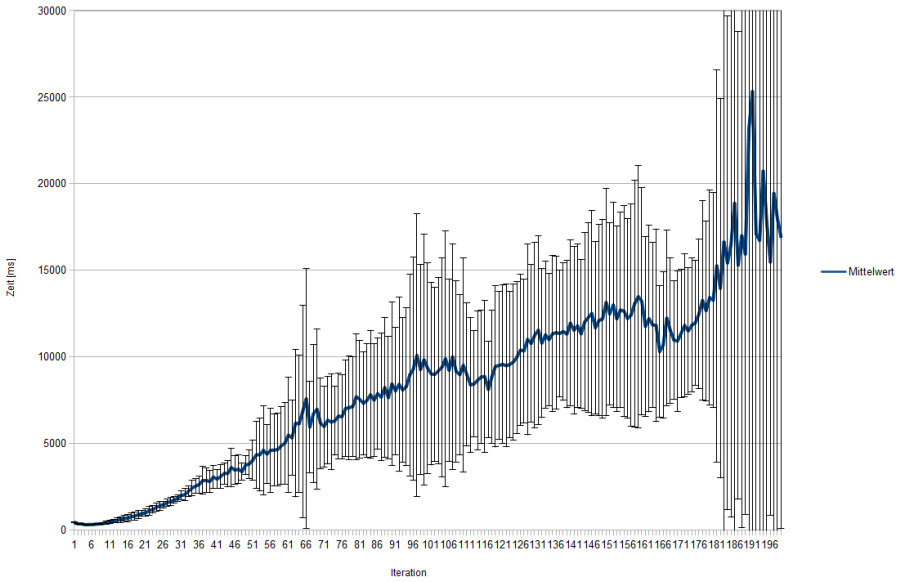

Abbildung A.7: Mittlere Dauer, Datensatz 2

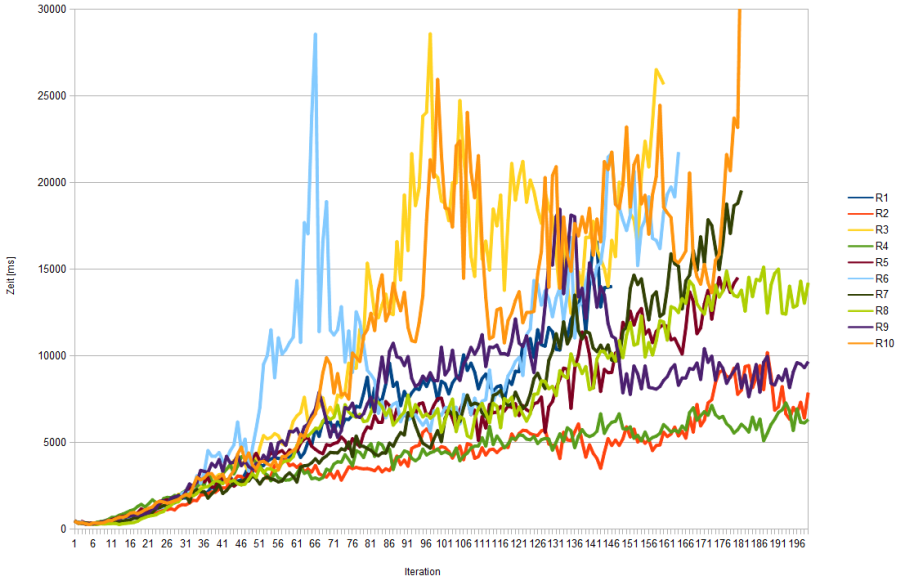

Abbildung A.8: Dauer, Datensatz 2