EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Mathematisch-Naturwissenschaftliche Fakultät

Technische Informatik

Bachelorarbeit

Bewertung Maschinelle Lernalgorithmen für die Güte von Programmierkenntnissen

Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Technische Informatik
Christian Hackenbeck, christian.hackenbeck@student.uni-tuebingen.de, 2020

Bearbeitungszeitraum: 01.01.2020-_____

Betreuer: Dr. Wolfgang Fuhl
Gutachter: Prof. Dr. Enkelejda Kasneci

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Christian Hackenbeck (Matrikelnummer 4120070), _____

Abstract

In dieser Arbeit soll untersucht werden, ob mithilfe von Eye-Tracking Daten und Maschinellen Lernalgorithmen eine Unterteilung von Probanden in verschiedene Expertise-Level (Novize und Experte) möglich ist. Hierbei wurden die Probanden mit einem Eye-Tracker bei der Bewältigung verschiedener Aufgaben aufgezeichnet. Probanden mit eindeutigen Fehlern bei der Bewältigung der Aufgabe wurden als Novize, Probanden mit unbedeutenden oder gar keinen Fehlern wurden als Experte eingestuft. Anschließend wurden aus diesen Daten Merkmale mit SubsMatch extrahiert und für Maschinelle Lernalgorithmen verwendet. Zum Vergleich dieser Auswertung mit SubsMatch wurde Multimatch zur Merkmalsextraktion genutzt. Das Ergebnis ist eine Unterteilung in Novize und Experte, auch wenn die gewonnenen Merkmale aufgrund der zu ähnlichen Expertise der Probanden zur Unterteilung nicht optimal sind.

Inhaltsverzeichnis

1. Einleitung

Das Verständnis von Programmcode benötigt einen enormen zeitlichen Aufwand und ist darüber hinaus meistens sehr fordernd. Programmierer, welche Software entwickeln, müssen den Code erst gänzlich verstehen, um Fehler in diesem zu finden. Es ist eine kognitive und komplexe Fähigkeit den Code in seinem Ablauf zu verstehen [Bla]. Seit Jahren wird durch Methoden und Techniken versucht, den Prozess des Verständisses zu entschlüsseln und zu verstehen. [BT06]

In jüngerer Vergangenheit wird dies durch verschiedene Studien in Zusammenarbeit mit Eye-Trackern versucht [BT06] [UNMiM06] [KRJ17]. Ein Ansatz, um dies besser zu verstehen, ist die Unterscheidung von Novize und Experte bei Programmierern. Crosby et al. [CSW02] versuchten dies noch ohne Eye-Tracker direkt am Code in der Benennung von aussagekräftigen Codezeilen. Hierbei gab es erste Indizien für einen Unterschied, da bei unterschiedlichen Gruppen verschiedene Codezeilen genannt worden sind. Uwano et al. [UNMiM06] verstärkten dies, indem Probanden mit Eye-Trackern beobachtet worden sind. Das Ergebnis war, dass eine längere Betrachtungszeit gut für das Erkennen von Fehlern ist. Somit konnten erste Parameter für weitere Studien gesetzt werden, welche sich auf Verständnis von Code spezialisierten.

Jüngst wurde dann von Chandrika und Amudha [KRJ17] in einem Experiment mit Eye-Trackern und Code-Beispielen gezeigt, dass Experten fehlerhaften Code wesentlich länger betrachteten als Novizen und erwartungsgemäß besser abschnitten. Eine Unterscheidung muss also möglich sein.

Um dies zu reproduzieren, wird in dieser Arbeit versucht, dies weiter mit Hilfe von Merkmalsextraktionen aus dem Scanpath und Maschinellem Lernen zu analysieren. Hierzu werden Probanden einem zweiteiligen Test unterzogen: Im ersten Teil müssen die Probanden erklären was bei dem präsentierten Code bei Ausführung geschieht. Im zweiten Teil müssen die Probanden in Code-Beispielen Fehler erkennen. Anschließend werden diese in zwei Personengruppen (Novizen und Experten) unterteilt. Nach der Unterteilung findet eine Merkmalsextraktion aus den Daten statt. Auf den extrahierten Merkmalen werden die Maschinellen Lernalgorithmen angewandt. Am Ende werden dann die Ergebnisse präsentiert, diskutiert und verglichen. Ein Ausblick auf weitere Schritte wird vorgestellt.

2. Grundlagen

In diesem Kapitel werden die Grundlagen für die spätere Studie erklärt. Diese Grundlagen umfassen das Eye-Tracking, die Merkmals-Extraktion und die auf den Daten der Merkmals-Extraktion genutzten Maschinellen Lernalgorithmen.

2.1. Eye-Tracking

Eye-Tracking ist das Messen der Blickposition von Lebewesen. Da jedes Lebewesen mit seinen Augen Aussagen darüber trifft, wo aktuell der visuelle Fokus liegt, werden somit umfassende Einblicke in die Gedankenwelt dieses Lebewesens gewonnen. In einigen Fällen kann man beispielsweise das Verständnis darüber erlangen, warum sich ein bestimmtes Verhalten gezeigt hat oder zeigt [KWK+13]. In der Forschung am Menschen fällt dies am Computer unter Mensch-Computer-Interaktion (HCI), welche sich mit der Schnittstelle Mensch-Maschine befasst [Car03]. Auf diesem Gebiet wird erforscht, wie Menschen mit Computern und (Design-)Technologien (versch. Interfaces) interagieren. Innerhalb der HCI gibt es weitere Teilgebiete, welche sich speziell mit dem Design [Car03], Verhalten des Menschen [GJG04], etc. beschäftigen und verschiedene Schwerpunkte legen. Eye-Tracking ist hierzu eine Möglichkeit, um diese Schnittstelle Mensch-Computer zu erforschen.

Beim Eye-Tracking geht man heutzutage folgendermaßen vor: Die Blickposition wird aus den Bewegungen des Auges berechnet. Das Auge vollzieht hierbei meist zwei Typen von Bewegungen, die man messen kann: Sakkaden und Fixationen. Sakkaden sind kurze, ruckartige Bewegungen des Auges, Fixationen sind kurze Fokusierungen beim Sehen. Die Fixationen können gemessen und aus diesen die aktuelle Blickposition bestimmt werden. In vielen Fällen werden diese Daten gespeichert und für eine Aufbereitung bzw. Evaluationen gesichert. Eine Auswertung erfolgt beispielsweise anschließend durch Bildung eines Verlaufs der Fixationen (Scanpath).

2.1.1. Das Auge

Die Grundlage im Eye-Tracking bildet das Auge. Es ist das Sehorgan des Menschen. Gesehen wird hierbei durch Einfallen von Licht im Auge, welches durch die Pupille und Linse strahlt. Dabei entsteht auf der Netzhaut ein Bild „auf dem Kopf“ von dem Gesehenen. Dieses Bild wird auf der Netzhaut mit Stäbchen und Zäpfchen wahrgenommen und anschließend über den Sehnerv an das Gehirn gesendet. Dieses verarbeitet es weiter und korrigiert bzw. ergänzt alle empfangenen Signale zu dem wahrgenommenen Bild [Web20].
Der Mensch sieht dabei immer nur einen kleinen Teil sehr scharf. Dies ist die Fovea Centralis, die Sehgrube im Zentrum des Gelben Flecks im Auge. Innerhalb dieses Gelben Flecks sind die Zäpfchen, welche der Farbwahrnehmung dienen, am höchsten konzentriert. Da hier die Sehschärfe am höchsten ist, bewegt der Mensch die Augen, um die Umgebung im Detail wahrzunehmen. Eine Verarbeitung von allem Unscharfen wäre ein enormer Aufwand für das Gehirn.

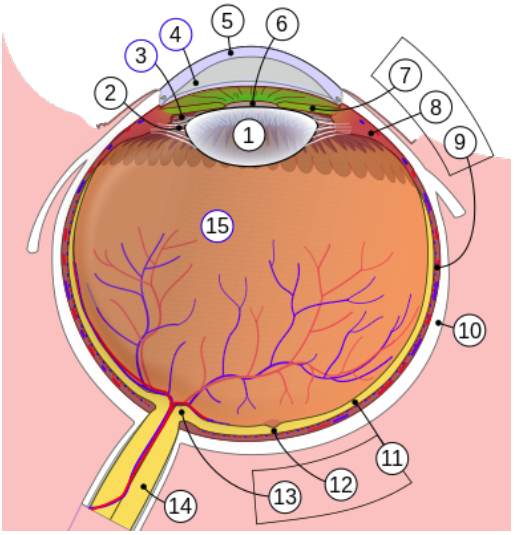


Abbildung 2.1.: Schematische Zeichnung des Auges
Abbildung entnommen aus [Wik20]

Im Folgenden wird noch auf den Aufbau des Auges eingegangen und einige Teile genauer beschrieben. Die Nummerierung bezieht sich hierbei auf Abbildung 2.1.

1. Linse – Bündelt das Licht für ein scharfes Bild auf der Netzhaut
2. Linsenaufhängeband
3. Hintere Augenkammer – Enthält Kammerwasser. Dies dient zur Desinfektion der anliegenden Teile des Auges
4. Vordere Augenkammer – Enthält Kammerwasser. Dies dient zur Desinfektion der anliegenden Teile des Auges
5. Hornhaut – Abschluss des Auges nach außen. Lichtdurchlässig
6. Pupille – Öffnung durch die das Licht ins Augeninnere gelangt
7. Iris – Trennung zwischen vorderen und hinteren Augenkammer. Innerer Rand mündet an die Pupille. Bildet die Blende des Auges
8. Ziliarkörper – Aufhängung und Akkommodation der Linse. An der Produktion

des Kammerwassers beteiligt.

9. Aderhaut
10. Sclera – Lederhaut des Auges; auch weiße Augenhaut genannt. Äußerste Schicht des Auges
11. Retina (Netzhaut) – Nervengewebe des Auges. Enthält die Nervenzellen zur Farb- bzw. Helligkeitswahrnehmung.
12. Fovea Centralis – Sehgrube (Vertiefung) mit den meisten Farbwahrnehmungszellen (Zäpfchen), daher beste Auflösung innerhalb dieses Punktes.
13. Sehnervenscheibe
14. Sehnerv – Verbindung an das Gehirn und übermittelt die aufgenommenen Daten des Auges
15. Glaskörper

Aufzählung gekürzt und entnommen aus [BS20]

2.1.2. Historischer Kontext

Ab dem 19. Jahrhundert wurde Eye-Tracking das erste Mal durchgeführt. Hierbei hatte der französische Augenarzt Emile Javal seine Patienten beim Lesen beobachtet und die Betrachtungsmuster der Patienten versucht zu beschreiben (beschrieben in [Hue08]). Erste Ansätze mit einem heute vergleichbaren Eye-Tracker wurden von Buswell [Bus22] durchgeführt. Mit Hilfe eines sich im Auge reflektierenden Lichtstrahles, der anschließend auf einem Film festgehalten worden ist, wurden die Augenbewegungen festgestellt. So konnte man diesen danach noch weiter händisch auswerten. Yarbus [Yar67] wendete 1967 die Eye-Tracking-Methode mit höherer Genauigkeit an und gilt daher häufig als erster Eye-Tracker: Mit Hilfe von am Auge angebrachten Vorrichtungen (siehe Abbildung 2.2) konnte so eine enorme Genauigkeit in der Analyse erreicht werden.

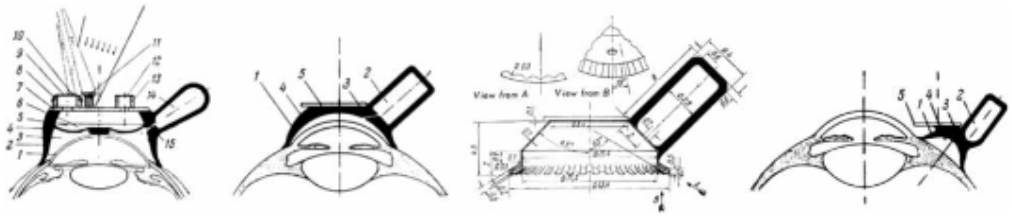


Abbildung 2.2.: Diagramm Saug-Kappen nach Yarbus. Abbildung ganz rechts zum Aufnehmen von Augen-Bewegungen
Abbildung entnommen aus [Yar67]

Dies war aber für den Probanden nicht immer angenehm, daher kam man von dieser Methode ab. Yarbus selber verwendete im späteren Verlauf seiner Forschung auch andere Methoden mit Kameras.
Ab den 70er Jahren entstanden die Eye-Tracker wie sie heute häufig beim Menschen in Gebrauch sind, welche eine höhere Genauigkeit erzielen und gleichzeitig für den Probanden angenehmere Versuchsdurchführung ermöglichen. Die höhere Genauigkeit ist unter anderem auf verschiedene Arten von Kameras (Eye-Tracking) und vor allen Dingen besseren Kameras zurückzuführen. Heutzutage existieren hauptsächlich zwei unterschiedliche Arten von Eye-Trackern: Head Mounted Eye-Tracker und die Remote Eye-Tracker.

2.1.3. Head Mounted Eye-Tracker

Beim Head Mounted Eye-Tracker wird der Eye-Tracker am Kopf des Probanden befestigt. Ein Head Mounted Eye-Tracker eignet sich besonders für dynamische Versuche, bei denen der Proband sich bewegen kann oder darf, da der Eye-Tracker sich mitbewegt. [SAB+18] Die Kamera ist meist in direkter Nähe zu den Augen befestigt und zeichnet meist auch nur die Augen bzw. das nähere Umfeld auf.

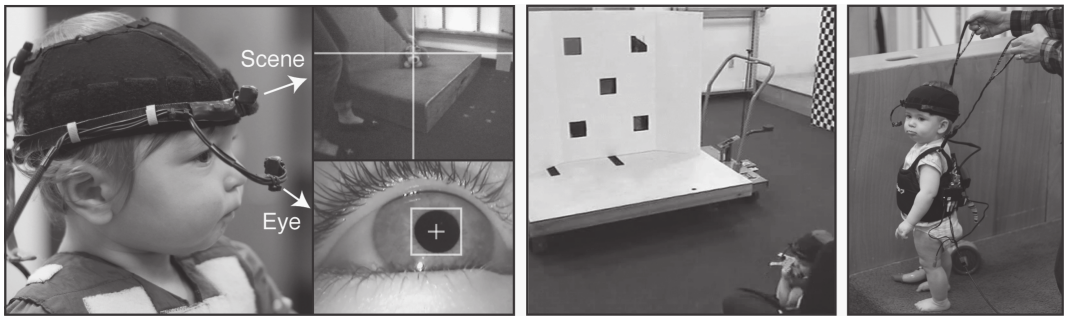


Abbildung 2.3.: Ein Head-Mounted Eye-Tracking Device am Kopf eines Babys befestigt, um zu verstehen wie das Baby seine Umgebung wahrnimmt / anschaut
Abbildung entnommen aus [Fra17]

2.1.4. Remote Eye-Tracker

Die zweite Variante ist der Remote Eye-Tracker. Hierbei ist der Eye-Tracker nicht am Probanden befestigt, sondern befindet sich in seinem direkten Blickfeld. Der Proband muss für gute Aufnahmen sehr still halten oder aber fixiert werden. Dieser Aufbau eignet sich daher nur, wenn der Proband sich kaum oder gar nicht bewegen muss. In dieser Studie wurde ein Remote Eye-Tracker verwendet.

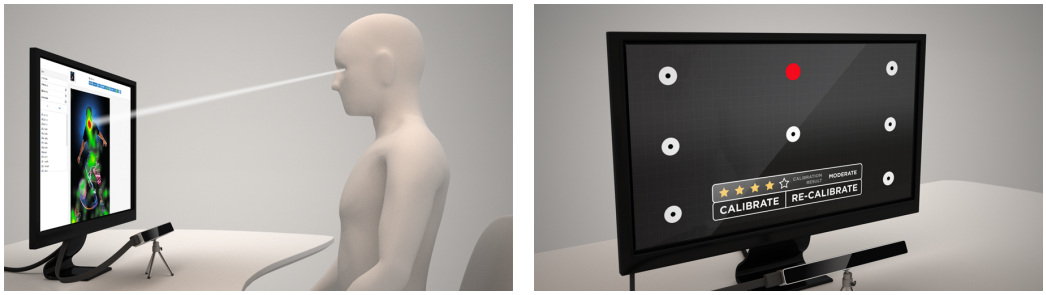
Abbildung 2.4.: Ein Remote Eye-Tracker genutzt, um bei einem Proband die Betrachtung eines Schach-Spielbretts nachzuvollziehen
Abbildung entnommen aus [Res20]

2.1.5. EyeTribe Eye-Tracker

Entwickelt wurde der verwendete Eye-Tracker von EyeTribe [Eye16a], einer dänischen Firma mit dem Ziel, die Eye-Tracker Technologie auch konsumentenfreundlich zu machen. Der Eye-Tracker von EyeTribe ist ein kleiner, kompakter Eye-Tracker, der nur über ein USB-Anschluss-Kabel verfügt. Aufgrund einer hohen Datenrate beim Messen der Blickposition ist eine USB-3.0 Schnittstelle von Nöten, um diese Daten zu übertragen. Diese bietet mehr Leistung und eine höhere Datenübertragungsrate. Aufgrund seiner Kompaktheit bietet der Eye-Tracker nur eine Abtastrate (Anzahl der Positionserfassungen) von 30Hz oder 60Hz. Andere Eye-Tracker können bis zu 2000Hz [Res20].
Weitere Eckdaten zum Eye-Tracker sind in Tabelle 2.1 aufgeführt.

Abtastrate	30 Hz und 60 Hz
Genauigkeit	0.5° (durchschnittlich)
Räumliche Auflösung	0.1°
Latenz	<20 ms bei 60 Hz
Kalibrierung	5, 9, 12 Punkte
Reichweite	45 cm – 75 cm
Wirkungsbereich	40 cm × 30 cm bei 65 cm Abstand
Bildschirmgröße	Bis zu 24 Zoll
Daten Output	Binokular Blick Daten

Tabelle 2.1.: EyeTribe Eye-Tracker Eckdaten



(a) Symbolbild von der Nutzung des EyeTribe EyeTrackers

(b) Kalibrierung nach Software Installation und Verbinden mit Eye-Tracker

Abbildung 2.5.: Verbildlichung des Einsatzes des EyeTribe Eye-Trackers
Abbildung entnommen aus [Eye16b]

In diesem Anwendungsfall wird auf dem Endgerät (z.B. Laptop) die von EyeTribe bereitgestellte Software installiert. Diese bildet das Interface auf dem Endgerät und bietet eine integrierte Option zur Kalibrierung des EyeTrackers. Nach erfolgter Kalibration wird ein Service gestartet, welcher einen TCP Server beinhaltet. Mittels eines TCP Clients können die generierten Daten nach erfolgtem Verbindungsaufbau zur EyeTribe Software empfangen und verarbeitet werden.

2.1.6. Anwendungsgebiet

Das Anwendungsgebiet von Eye-Tracking ist sehr vielseitig. Genutzt werden kann es bei Usability-Tests von Software, wobei die Nutzerfreundlichkeit eines Interfaces ermittelt und dies unter anderem mit Eye-Tracking realisiert wird [GJG04]. Auch in den Neurowissenschaften in Zusammenarbeit mit einem EEG [KWK+13], in der Psychologie bei der Wahrnehmungsforschung [SAB+18] oder in der Marktforschung für neue Produkte/Werbungen [KWK+13] wird Eye-Tracking zur Messung und anschließender Bewertung genutzt. In der HCI findet es beispielsweise in der Analyse von Code statt [CAS17].

2.1.7. Code Analyse

Bei der Code Analyse wird der Proband beobachtet, wie er an bestimmte Aufgaben herantritt [CAS17]. Dabei kann die Aufgabe gelöst, verbessert oder nur analysiert werden. Dies geschieht entweder via Eingabe des Probanden oder durch lautes Denken. Die Eingabe stellt bei einem Remote Eye-Tracker jedoch ein Problem dar, solange der Proband nicht fixiert ist, weil das Schreiben auf der Tastatur zwangsweise das Abwenden vom Monitor und somit vom Eye-Tracker zur Folge hat. Um diese Fehlerquelle zu vermeiden, wird daher in dieser Arbeit der Code vom Probanden analysiert und mit der Methode des lauten Denkens korrigiert. Hierbei

wird festgehalten, wie genau der Code vom Probanden erklärt wird und ob die Erklärung verständlich ist.

2.2. Merkmals-Extraktion

Im Folgenden werden die zwei genutzten Algorithmen, SubsMatch und Multimatch, zur Extraktion der Merkmale aus den Daten erklärt. Dabei wird darauf eingegangen, wie diese funktionieren und angewendet werden.

2.2.1. SubsMatch 2.0

Ein Algorithmus zur Merkmals-Extraktion innerhalb der Daten ist SubsMatch 2.0 [KRS+16]. SubsMatch 2.0 ist hierbei eine Verbesserung von SubsMatch [KKR14], bei welchem die Fixationen und Sakkaden als Grundlage für einen Scanpath genommen werden. Im Weiteren wird SubsMatch 2.0 als SubsMatch bezeichnet. Ein Scanpath besteht aus einer Verkettung von Fixationen und bildet somit den Betrachtungsverlauf des Probanden.
Dieser Scanpath wird in einen String umgewandelt und basiert auf Position und Dauer der aufgezeichneten Fixationen. Da die Auflösung eines Bildschirms zu groß ist und manuell eingestellte statische Areas-of-Interest (AOI) zu aufwendig für einen reellen Einsatz sind, verwendet SubsMatch für die Positionsbestimmung eine Unterteilung des Bildschirms in gleichmäßige Felder (ein sogenanntes Grid). Die Größe des Grids ist einstellbar und teilt den Bildschirm gleichermaßen in Felder auf. Jedes Feld in diesem Grid erhält einen Buchstaben. Die Aufenthaltsdauer in einem Feld des Grids wird im String durch eine Wiederholung des jeweiligen Grid-Namens (Buchstabens) ausgedrückt [KRS+16]. Der so entstandene String ist eine lange Sequenz aus Buchstaben.
SubsMatch versucht nun in einem zweiten Schritt innerhalb dieses Strings Subsequenzen einer bestimmten Länge zu erkennen. Ähnliche bzw. gleiche Subsequenzen deuten hierbei auf eine Ähnlichkeit zwischen verschiedenen Scanpaths hin und können somit zur Klassifikation genutzt werden.

2.2.2. Multimatch

Multimatch [DNJ+12] ist der zweite Algorithmus zur Merkmals-Extraktion. Dieser verarbeitet die Positionsdaten und die Dauer der Fixationen und berechnet daraus Vektoren im Raum zwischen den einzelnen Fixationspunkten. Diese Vektoren zwischen den einzelnen Fixationen bilden in der Abfolge einen Scanpath. Innerhalb dieses Scanpaths lassen sich Vergleiche in 5 Dimensionen ausführen (siehe Tabelle 2.2).

Form: Vektor-Differenz zwischen zwei verknüpften Vektoren. Hierbei werden die Endpunkte der Vektoren verglichen. Dies dient zum Vergleich der Scanpath Form.	
Länge: Vergleich der Länge der Vektoren	
Richtung: Winkelvergleich zwischen Vektoren, wenn die Länge unterschiedlich ist.	
Position: Vergleich der Positionen von Vektoren.	
Dauer: Vergleich der Dauer des Vektors.	

Tabelle 2.2.: die 5 Dimensionen visualisiert
Tabelle entnommen und übersetzt aus [DNJ+12]

Anschließend werden diese Pfade verglichen, um Ähnlichkeiten zwischen verschiedenen Beobachtungen zu ermitteln. Das Besondere an diesem Ansatz ist die räumliche Unabhängigkeit und die Skalierbarkeit: Es ist möglich eine Verschiebung der Fixationen bzw. eine Skalierung (Verkleinerung/Vergrößerung) dieser zu erkennen und als ähnlichen Scanpaths wahrzunehmen. Diese Ähnlichkeitsermittlung geschieht durch den unterschiedlichen Einsatz der 5 Dimensionen in Tabelle 2.2. Anders als SubsMatch gibt Multimatch direkt einen Vergleichswert des Inputs an und benötigt keine weiteren Maschinellen Lernalgorithmen.

2.3. Maschinelles Lernen

Maschinelles Lernen (ML) bezeichnet den Vorgang der Generierung von Wissen aus Daten bei Maschinen. Das Wissen setzt sich aus gesammelter Erfahrung zusammen, welche aus zugrundeliegenden Daten gewonnen wird. Innerhalb dieser Arbeit werden folgende ML-Algorithmen aus Matlab [Mat20a] verwendet.

- Entscheidungsbäume
  - Feiner Baum
  - Medium Baum
  - Grober Baum
- Diskriminanten Analyse
  - Linear
  - Quadratisch
- Naiver Bayes Klassifizierer
  - Gaußischer Naiv Bayes
  - Kernel Naiv Bayes
- Support Vektor Maschinen (SVM)
  - Linear
  - Quadratisch
  - Kubisch
  - Feiner Gaußischer SVM
  - Mittlerer Gaußischer SVM
  - Grober Gaußischer SVM
- Nächste Nachbarn Klassifizierer (kNN)
  - Feiner kNN
  - Medium kNN
  - Grober kNN
  - Kosinus kNN
  - Kubischer kNN
  - Gewichteter kNN

- Ensemble Klassifizierer
  - verstärkte Entscheidungsbäume
  - verschachtelte Entscheidungsbäume
  - Subspace Diskriminante
  - Subspace kNN
  - RUS-verstärkte Entscheidungsbäume

Entscheidungsbäume [Mat20b]: Entscheidungsbäume, oder Klassifizierungsbäume, geben eine Vorhersage (Antwort) für die zugrunde liegenden Daten. Um diese Vorhersage zu treffen, folgen sie den Entscheidungen (Wahrscheinlichkeiten) innerhalb des Baums von der Wurzel hin zu den Blättern. Die Entscheidung wird hierbei in den Knoten getroffen. Der Knoten enthält zwei Prädikatoren und die Wahrscheinlichkeit für diese Prädikatoren. Je nachdem welche Wahrscheinlichkeit bei welchem Prädikator höher ist, wird im Baum Richtung Blatt hinabgestiegen. Nach endlich vielen Iterationen dieses Vorgehens steht im Blatt die binäre Antwort. Klassifizierungsbäume geben hierbei binär *wahr* oder *falsch*, Regressionsbäume hingegen geben numerische Antworten.
Diskriminanten Analyse [Mat20c]: Die Diskriminanten Analyse nimmt an, dass verschiedene Klassen Daten generieren basierend auf einer Gaußschen Verteilung. Dabei wird jeder Klasse Y eine gaußsche Verteilung X basierend auf den zugrunde liegenden Daten (Parameter) zugeordnet. Anschließend wird mit diesen verschiedenen Klassen Y_i verglichen und die Klasse mit der höchsten Übereinstimmung für die gegebenen Parameter, unter Berücksichtigung der Wahrscheinlichkeit und der Fehlerklassifizierungskosten, ausgewählt.
Naiver Bayes Klassifizierer [Mat20d]: Der Naiver Bayes Klassifizierer ist für unabhängige Merkmale innerhalb einer Klasse erstellt worden. Er klassifiziert in zwei Schritten: Zuerst trainiert er auf Trainingsdaten (Trainingsschritt) und versucht Parameter dieser Daten auszulesen. Anschließend nutzt er das gewonnene Wissen aus diesen Parametern im Vorhersage-Schritt, um für die zu klassifizierenden Daten Aussagen über deren Klassenzugehörigkeit zu treffen. Hierbei vergleicht er sein gewonnenes Wissen (Parameter) mit den ermittelten Parametern aus den zu klassifizierenden Daten und trifft dann seine Entscheidung.
Support Vektor Maschinen (SVM) [Mat20e]: Die Support Vektor Maschinen arbeiten auf der Basis, die Daten in Klassen mit Hyperebenen im Raum zu trennen und so eine möglichst gute Klassifizierung der Daten zu ermöglichen. Dabei wird jeder Datenpunkt mit Klassenzugehörigkeit (Label) in einem n-dimensionalen Raum dargestellt. Anschließend werden Hyperebenen gebildet, um diese Gruppen möglichst gut zu umschließen und einzuteilen. Die Platzierung der Hyperebene wird mit einem möglichst großen Abstand zu dem nächstmöglichen Datenpunkt durchgeführt.
Nächste Nachbarn Klassifizierer (kNN) [Mat20f]: kNN schätzt aufgrund der nächsten Nachbarn des Datenpunktes die Klasse des betrachtenden Datenpunktes ein. Hierbei kann nach verschiedenen Werten, wie der Anzahl der zu untersuchenden

Nachbarn, vorgegangen werden. Dies funktioniert wie folgt: Der zu untersuchende Datenpunkt wird zu einer Klasse X zugehörig befunden, wenn die Mehrheit der k nächsten Nachbarn dieser Klasse X angehören. k muss nicht unbedingt eine Zahl von Nachbarn sein, sondern kann auch ein Radius (für eine Sphäre) um den Datenpunkt herum darstellen.
Ensemble Klassifizierer [Mat20g]: Ein Ensemble Klassifizierer vereint verschiedene Vorhersagemethoden und gewichtet diese unterschiedlich. Allgemein gehalten ermöglicht dies eine bessere Vorhersage für die zugrundeliegenden Daten. Die vielen schwächeren kleinen Learner sind kombiniert besser im Klassifizieren als einzelne Learner. Unter anderem sind diese Learner Entscheidungsbäume. Zu unterscheiden sind diese in *Boosted* (verstärkte) und *Bagged* (verschachtelte) Entscheidungsbäume: *Boosted* sind flache Bäume mit genau der Tiefe 1 und einer Entscheidung, *Bagged* hingegen sind die Entscheidungsbäume mit größerer Tiefe.

3. Verwandte Arbeiten

Bereits 2002 fanden Crosby et al. [CSW02] heraus, dass es Unterschiede im Verständnis von Code gibt. Dies wurde mithilfe von verschiedenen stereotypischen, sehr aussagekräftigen Code-Zeilen umgesetzt. Diese Zeilen wurden dahingehend als *Beacons* bezeichnet, da sie im Verständnis des Codes wegweisend sind (beacon engl. für "Leuchtturm"). Hierbei wurden verschiedene Probanden eingesetzt, die beurteilen mussten, welche Zeilen Code für sie *Beacons* darstellen. Unterschiede ergaben sich hier in der Benennung der *Beacons* zwischen Novizen und Experten, wobei Novizen teilweise gar keine Zeilen nannten.
Daraufhin wurden im Jahre 2006 von Uwano et al. [UNMiM06] mit Hilfe eines Eye-Trackers weitere Untersuchungen in Richtung Güte-Bestimmung durchgeführt. Hierfür wurden (ähnlich zu Crosby et al. [CSW02]) Probanden dazu aufgefordert, Code zu analysieren. Dabei wurden die Probanden mit dem Eye-Tracker beobachtet und untersucht, mit welchem Blickmuster diese versuchen ihn zu verstehen. Dabei wurde mit Hilfe der gemessenen Positionsdaten die aktuelle Codezeile bestimmt und anschließend ausgewertet. Ein Verlauf innerhalb der aufgenommenen Positionsdaten wurde zwar auch beachtet, jedoch wurde mehr die zugrunde liegende Beobachtungszeit berücksichtigt. Probanden mit einer längeren Beobachtungszeit des Codes fanden hierbei häufiger den Fehler im Code, als Probanden, die den Code über einen kürzeren Zeitraum betrachtet hatten. (Siehe Abbildung 3.1)

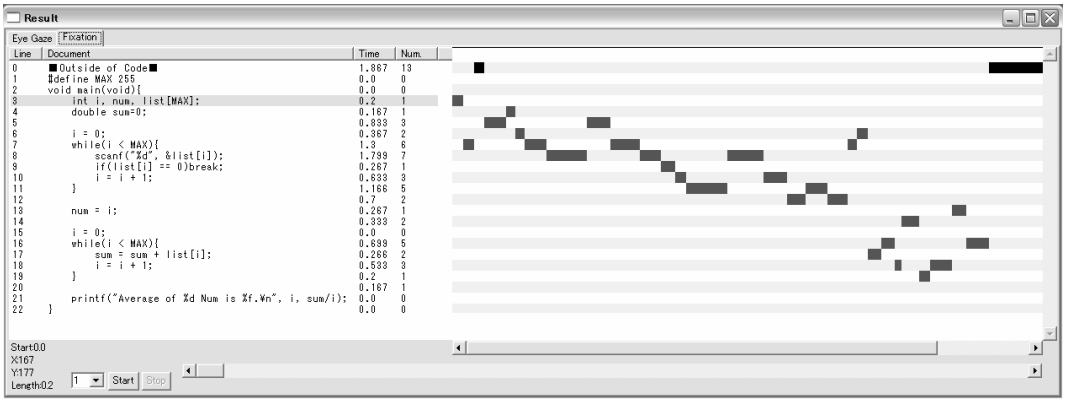


Abbildung 3.1.: Links zu sehen der Code und rechts der zeitliche Verlauf wie lange auf einer Codezeile geschaut wurde
Abbildung entnommen aus [UNMiM06]

Diese Erkenntnis wurde von Bednarik und Tukiainen [BT06] weiter ausgeführt. Dabei wurde der Code auf dem Bildschirm in einem Tool zur Visualisierung dargestellt. Dieses Graphical User Interface (=GUI) teilte den Bildschirm in hauptsächlich zwei große Regionen ein: Links der Code, welcher von Bednarik und Tukiainen vorgegeben wurde und rechts eine Visualisierung des Codes. Die Visualisierung wurde unterteilt in Methoden-Aufrufe, Auswertungen, Konstanten bzw. statische Objekte und Objekte bzw. Arrays. Diese visuelle Darstellung verschaffte dem Betrachter einen guten Überblick über den Code auf der linken Seite. (Siehe Abbildung 3.2)

Dabei kam heraus, dass Novizen viel schneller auf die Visualisierung schauen, als dass sie versuchen, den Code ohne Hilfsmittel zu verstehen. Experten hingegen betrachteten länger den linken Teil mit Code, um diesen zu verstehen.
Sharif et al. [SFM12] gingen einen Schritt weiter als Uwanao et al. und erweiterten den Versuchsaufbau, indem sie eine größere Zahl an Probanden einsetzten. Dabei wurde explizit auf das Suchen von fehlerhaftem Code und die damit verbundene Zeit betrachtet. Erneut kam heraus, dass eine längere Lesezeit des Codes förderlich für das Finden von Fehlern war.
Busjahn et al [BSB11] untersuchten, auf welchem Teil des Codes die Aufmerksamkeit beim Betrachten von Code liegt. Besonders betrachtet wurden hierbei die Literale, Operatoren und Schlüsselworte.

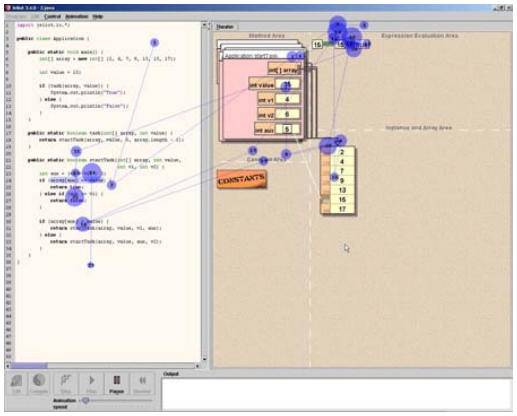
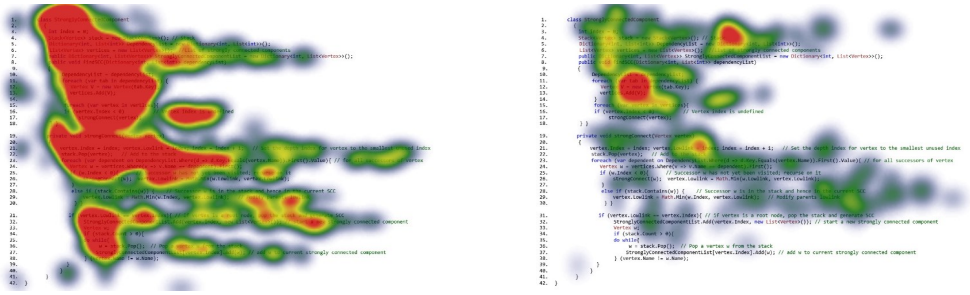


Abbildung 3.2.: Visualisierungs-Tool von Bendarik et al. mit eingezeichnetem Scanpath eines Probanden. Die Größe der Punkte ist relativ zu der Dauer der Betrachtung.
Abbildung entnommen aus [BT06]

Den Ansatz mit einer GUI verfolgten Hejady und Narayanan [HN12] weiter und versuchten mittels IDE zu testen, welche Präferenzen zum Verständnis von Code und anschließendem Debuggen des Codes von den einzelnen Expertise Gruppen bevorzugt genutzt werden. Sie generierten ähnlich wie Bednarik und Tukiainen für den Betrachter eine statische (z.B. UML Diagramm) und eine dynamische (z.B. visuelle Realtime Darstellung der Datenstruktur) Repräsentation des Codes. Hierbei fiel ihnen auf, dass der dynamische Teil wesentlich häufiger zum Debuggen bzw. Verstehen des Codes genutzt wurde als der statische Teil.
Weiteres zum Thema Code-Verständnis untersuchten Binkley et al. [BDL+12] mit dem Schwerpunkt von unterschiedlicher Variablen-Benennung. Der zu untersuchende Code wies dabei verschiedene Arten von Variablen-Benennungen auf: Binnenmajuskel (z.B. OnkelOtto - Zusammenschreibung mehrerer Wörter und

Großschreibung am Anfang des Wortes) und Unterstrich (z.B. Onkel_Otto). Die Studie zeigte einerseits die unterschiedliche Wahrnehmung von natürlichem Text und Code, andererseits konnte man auch feststellen, dass Novizen ein besseres Code-Verständnis bei Code mit Binnenmajuskel aufwiesen als Code mit Unterstrich. Bei erfahrenen Programmierern hingegen wurde kein Unterschied festgestellt.
Turner et al. [TFSL14] verglich die zwei Sprachen C++ und Python bezüglich dem Verständnis und Fehlererkennung. Zwischen diesen Sprachen traten keine Unterschiede auf, jedoch wurden fehlerhafte Code-Zeilen im Vergleich wesentlich länger betrachtet als fehlerfreie.
Chandrika und Amudha [KRJ17] untersuchten den Unterschied bei der Betrachtung von Code zwischen Novizen und Experten. Das Ergebnis zeigte, dass Experten den Fokus verstärkt auf fehlerhafte Code-Zeilen legten und erwartungsgemäß besser abschnitten als Novizen. Weiterhin untersuchten Chandrika et al. [CAS17] wie verschiedene Personen durch den Code gingen und ob Unterschiede basierend auf dem Können des Betrachters bestanden. Hierzu wurden Personen ohne Programmierkenntnisse und Personen mit guten Programmierkenntnissen verglichen. Innerhalb dieser Personengruppen wiesen Personen mit mehr Erfahrung eine höhere Abdeckung des Codes und eine längere Betrachtungszeit als Personen ohne Erfahrung auf. (siehe Abbildung 3.3)



(a) Heatmap einer Person mit Programmierkenntnissen

(b) Heatmap einer Person ohne Programmierkenntnisse

Abbildung 3.3.: Links eine Person mit viel Erfahrung. Eindeutig längere Betrachtung, da tiefer rot gefärbt. Auf der rechten Seite eine Person ohne Programmierkenntnisse. Eine geringere Code-Abdeckung wird durch viele freie Stellen und wenig Rotfärbung ausgedrückt.
Abbildung entnommen aus [CAS17]

4. Studie

In der folgenden Studie soll untersucht werden, ob die Güte von Programmierkenntnissen aus Eye-Tracking Daten extrahiert und mit ML klassifiziert werden kann. Untersucht wird dies mithilfe eines Eye-Trackers, verschiedenen Aufgaben in Form von Code-Verständnis und der anschließenden Auswertung durch Maschinelles Lernen. Die Probanden müssen hierzu verschiedene Aufgaben verstehen und erklären können. Die Aufgaben teilen sich hierbei in 4 Verständnis-Aufgaben und 4 Fehlersuch-Aufgaben auf. Während die Probanden diese Aufgaben lösen, werden sie via Eye-Tracker gemessen. Später erfolgt die Auswertung über SubsMatch und Multimatch (weitere Informationen zu den genutzten Algorithmen in Kapitel 2.3).

4.1. Vorbereitung

Die Vorbereitung ist gegliedert in die Erstellung der Aufgaben, die Beschreibung des Eye-Trackers, die Erstellung eines Clienten zum Empfangen von Daten und der Aufstellung eines Fragebogens.

4.1.1. Code Aufgaben

Ein Problem bei Aufgaben (Code) Auswahl ist die unterschiedliche Herangehensweise der Programmierer. Für einen Programmierer kann es aussehen, wie ein logischer Fehler im Code, der Urheber des Codes hat dies aber genauso gewollt, obwohl es nicht optimal/logisch ist. Die unterschiedliche Einrückung des Codes kann beispielsweise von vielen als Fehler angesehen werden, aber eine Intention des Erstellers sein, da er dies so besser lesen kann. Dies hat in den meisten Sprachen keinen Einfluss auf die Effizienz des Programms, fällt aber dennoch ins Auge. Daher wurden die Aufgaben zum Großteil mit Syntax-Fehlern (z.B. in Java ein Semikolon falsch gesetzt) erstellt. Darüber hinaus ist die Länge des Codes entscheidend: Ein Scrollen des Codes während der Durchführung würde eine geringere Validität der Messergebnisse bedeuten, da einige Probanden wahrscheinlich mehr/weniger scrollen würden als andere. Die erstellten Aufgaben bestehen daher aus wenigen Codezeilen (ca. 50 Zeilen), um einen guten Überblick zu gewährleisten und eine statische Umgebung zu schaffen.

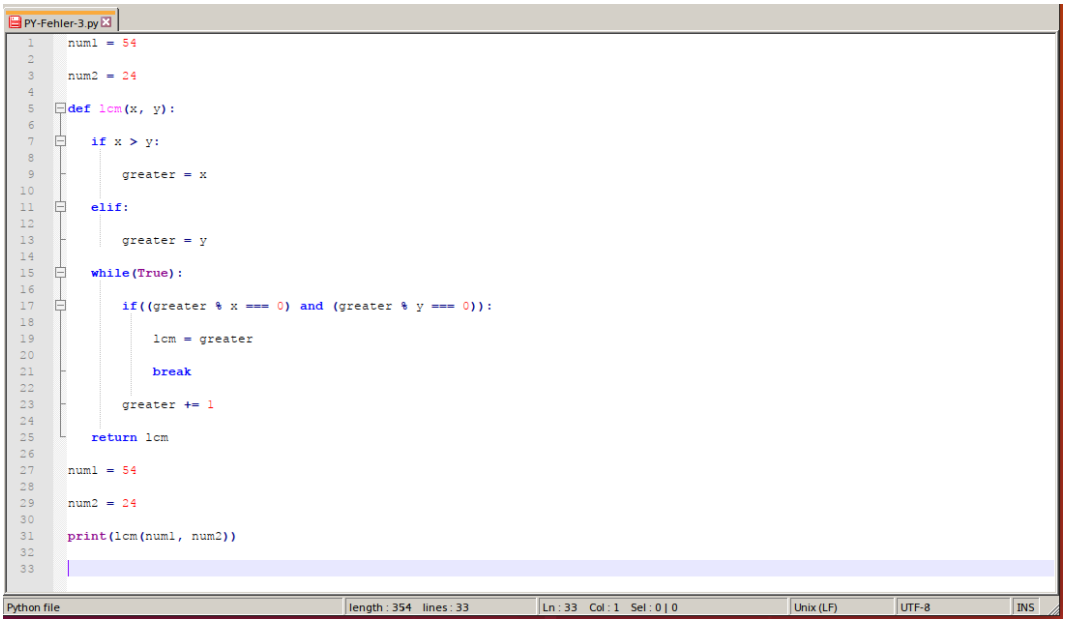
Die nächste Hürde ist die Schwierigkeit der Aufgaben: Jeder sollte sie verstehen können, aber nicht auf den ersten Blick lösen. Daher wurde bei der Erstellung der Aufgaben Wert darauf gelegt, dass diese verhältnismäßig einfach, aber dennoch anspruchsvoll sind. Ein zu einfaches Beispiel wäre eine Addition, ein zu schweres Beispiel die Berechnung des chinesischen Restsatzes zweier Zahlen.
Außerdem sollten die Aufgaben in möglichst allen Sprachen gleich sein und daher darf keine sprachspezifische Eigenheit, welche nur diese Sprache aufweist, vorkommen. Dies wurde mit der Ausnahme berücksichtigt, dass die Eingabe bzw. Ausgabe von Daten in einigen Sprachen anders gestaltet ist.
Unter Berücksichtigung dieser Aspekte wurde ein Aufgabenpool mit ca. 20 verschiedenen Aufgaben erstellt, welcher anschließend gefiltert worden ist. Ausgewählt wurden Aufgaben rund um den größten gemeinsamen Teiler (ggT) oder das kleinste gemeinsame Vielfache (kgV) neben anderen Aufgaben wie einem Schaltjahr-Rechner. All diese Aufgaben sind nicht allzu schwer, wenn man einmal den Code versucht zu verstehen, jedoch auch nicht auf den ersten Blick zu lösen. Dies ist eine gute Voraussetzung für eine anspruchsvolle, aber nicht überfordernde Aufgabe (beispielhaft dargestellt in Abbildung 4.1).

4.1.2. EyeTribe Eye-Tracker

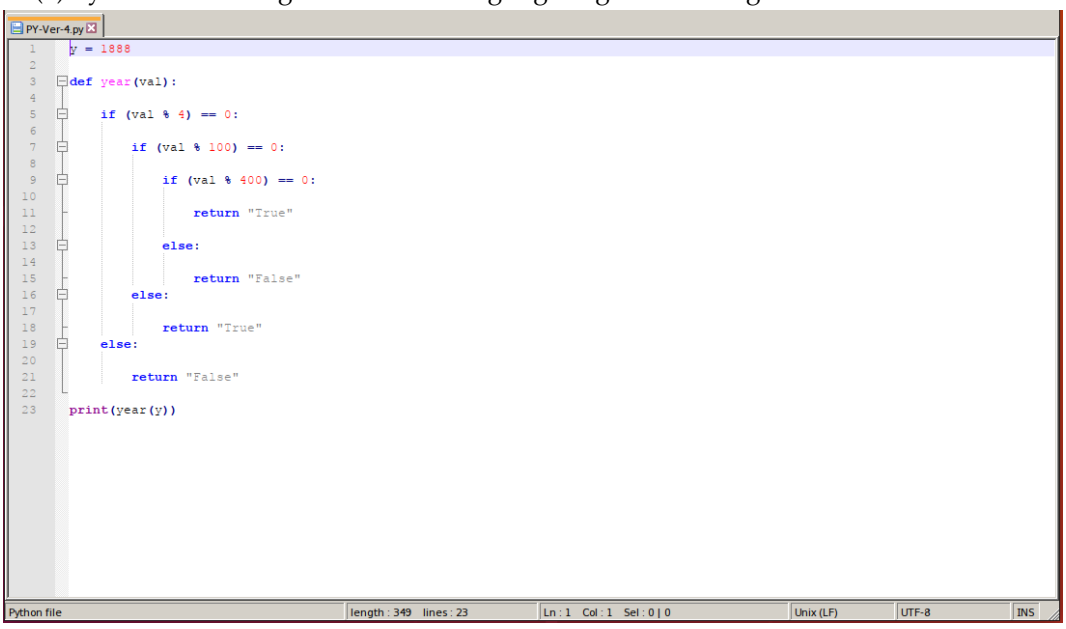
Die Datenaufnahme erfolgt mit dem Eye-Tracker von EyeTribe. Dieser Eye-Tracker erfasst 30-mal in der Sekunde die Blickposition des Probanden und sendet diese über das angeschlossene USB-Kabel an ein Endgerät. Auf der Seite des Endgerätes wird die EyeTribe Software betrieben, welche die Daten entgegen nimmt. Diese EyeTribe Software bietet einen Server an, welcher nach Verbindungsaufbau die Daten via TCP an einen Clienten sendet. Der Client wiederum kann diese Daten beliebig bearbeiten, speichern oder weiterreichen. Damit die Verbindung bestehen bleibt, muss der Client auf dem Endgerät einen Heartbeat (ein sich alle t Zeiten wiederholendes Signal) senden.
Auf diesen Clienten wird im folgenden Abschnitt weiter eingegangen.

4.1.3. Client für EyeTribe Eye-Tracker

Für diese TCP Schnittstelle am Endgerät wurde ein Client in C# geschrieben, welcher die Verbindung aufbaut, den Heartbeat sendet und anschließend die empfangenen Rohdaten als String abspeichert. Es wurden die Rohdaten gewählt, da so eventuell weitere Informationen neben Augenposition und Zeit extrahiert werden können. Dieser Client wurde darüber hinaus so ausgestattet, dass jede Aufgabe eines Probanden in eine eigenen Datei abgespeichert werden kann. Dies ermöglicht es bei den verschiedenen Aufgaben der Probanden je einen konkreten Abschnitt zu erstellen und später nicht die einzelnen Aufgaben auseinanderhalten zu müssen.



(a) Python-Fehleraufgabe mit Code-Highlighting - Kleinstes gemeinsames Vielfaches



(b) Python-Verständnisaufgabe mit Code-Highlighting - Schaltjahr-Rechner

Abbildung 4.1.: Zwei Python Aufgaben dargestellt in Notepad++

Je durchgeführte Aufgabe des Probanden entstand eine .txt-Datei. In Abbildung 4.2 ist der Code des Verbindungsaufbaus dargestellt. Die Methode CONNECT nimmt hierzu einen Host und einen Port entgegen und versucht eine Verbindung auf diesen Host mit angegebenem Port aufzubauen. Nach Verbindungsaufbau wird direkt der Heartbeat gesendet und ein Timer konfiguriert, welcher diesen alle 250 Millisekunden sendet. Über den Thread INCOMINGTHREAD wird ein Thread gestartet mit einem Loop zum Abhören (eine WHILE-Schleife die als Bedingung TRUE besitzt). Innerhalb dieses Loops werden die empfangenen Daten verarbeitet.

```
public bool Connect(string host, int port)
{
    try
    {
        socket = new TcpClient(host, port);
    }
    catch (Exception ex)
    {
        Console.Out.WriteLine("Error connecting: " + ex.Message);
        return false;
    }

    // Send the obligatory connect request message
    string REQ_CONNECT = "{\"values\":{\"push\":true,\"version\":1},\"category\":\"tracker\",\"request\":\"set\"}";
    Send(REQ_CONNECT);

    // Lauch a seperate thread to parse incoming data
    incomingThread = new Thread(ListenerLoop);
    incomingThread.Start();

    // Start a timer that sends a heartbeat every 250ms.
    // The minimum interval required by the server can be read out
    // in the response to the initial connect request.

    string REQ_HEATBEAT = "{\"category\":\"heartbeat\",\"request\":null}";
    timerHeartbeat = new System.Timers.Timer(250);
    timerHeartbeat.Elapsed += delegate { Send(REQ_HEATBEAT); };
    timerHeartbeat.Start();

    return true;
}
```

Abbildung 4.2.: Code vom Verbindungsaufbau des Clients

4.1.4. Fragebogen

Für die Selbsteinschätzung der Probanden und für die anschließende Auswertung dieser wurde ein Fragebogen erstellt. In diesem Fragebogen finden sich folgende Fragestellungen und Teilgebiete:

1. Seit wie vielen Jahren produzieren / schreiben Sie schon Code?
2. Wie viele Jahre arbeiten Sie mit Eye-Tracking? (0 falls noch gar nicht)
3. Wie alt sind Sie?
4. Benötigen Sie eine Sehhilfe?
5. Eine 10-teilige Likert-Skala zur Selbsteinschätzung der eigenen Erfahrungen

6. Wie viele Monitore nutzen Sie für gewöhnlich?
7. Wie groß sind diese Monitore?

In der Auswertung werden diese Daten mit den gemessenen Werten durch den Eye-Tracker verglichen.

4.2. Aufbau

Der Versuchsaufbau, bestehend aus einem externen Bildschirm, dem Eye-Tracker (von EyeTribe) und dem Laptop wurden auf dem Tisch aufgebaut. Der Bildschirm des Laptops wurde dabei um den externen Bildschirm erweitert und konnte von den Probanden nicht eingesehen werden. Der Bildschirm für die Probanden besaß eine Auflösung von 1920x1200px. Die Aufgaben wurden auf ein Maximales innerhalb des Editors gestreckt um den Bildschirm auszufüllen. Genutzt wurde Notepad++ [Not19] aufgrund von Highlights (Highlights im Code; siehe Abbildung 4.1) und einer weiten Verbreitung.

4.3. Durchführung

Alle Probanden wurden vom Lehrstuhl HCI der Universität Tübingen angeworben; die Teilnahme erfolgte freiwillig und ohne Entgelt.
Ein jeder Proband musste am Anfang den Fragebogen ausfüllen und eine Einverständniserklärung unterschreiben. Nach ausgefülltem Fragebogen durfte der Proband die Sprache auswählen. Die Aufgaben sind in Python, C++ und Java geschrieben. Ausgewählt wurden diese Programmiersprachen aufgrund der hohen Verbreitung. Ermittelt wurden sie über die Anfragenfrequentierung der großen Suchmaschinen wie Google, Bing und co. Nach Auswahl der Sprache wurde der Eye-Tracker auf den Probanden kalibriert. Dazu nahm der Proband auf dem Stuhl ca. 70cm vom Bildschirm entfernt Platz und die eingebaute Kalibrierung von EyeTribe wurde durchgeführt. Der Eye-Tracker wurde dabei am unteren Bildschirm-Rand platziert. Die Aufgaben bestanden aus 4 Verständnis- und 4 Fehlersuch-Aufgaben, wobei es dem Probanden frei stand, wie die Aufgaben bewältigt wurden: Insofern dieser den Code im Stillen überblicken und direkt verstehen konnte, durfte er dies tun und direkt sagen, was dieser Code macht bzw. was der Output dieses Codes ist. Falls nicht, musste er den Code deskriptiv angehen und laut denken. In allen Fällen wurde kein Zeitlimit für die Betrachtung des Codes angelegt. Alle Probanden schafften es jedoch unter 10 Minuten mit allen 8 Aufgaben fertig zu werden.
Bei den Fehleraufgaben wurde nur bekannt gegeben, dass die Aufgaben Fehler enthalten, jedoch nicht wie viele. Die Fehler beschränkten sich hierbei auf Fehler in der Syntax, da Logik-Fehler im Code nicht unbedingt Logik-Fehler sein müssen, sondern gewollt anders ausgedrückt sind (siehe hierzu auch 4.1.1 Code Aufgaben).

Zum Abschluss wurde jedem Probanden noch erklärt, worum es bei dem Versuch ging und es durfte Feedback geäußert werden.

5. Ergebnisse und Methoden

Die erfassten Daten aus Kapitel 4. Studie wurden für die weitere Verarbeitung mit den Extraktions- und Maschinellen Lernalgorithmen aufbereitet. Die Aufbereitung der Rohdaten erfolgte in einem eigenen Programm. Dieses Programm ermöglicht verschiedene Handhabungen der Rohdaten. Hier wurde ein Export im Format $X;Y;t$ gewählt. X und Y sind Pixel-Koordinaten und t ist der aktuelle Zeitpunkt. Anschließend wurden die Daten in Matlab [Mat20a] importiert und dort weiterverarbeitet. In Matlab erfolgte die Extraktion der Merkmale mit SubsMatch. Die Auswertung von Multimatch erfolgte auf einem separaten Linux-System als Python Skript. Das Kapitel ist unterteilt in Aufbereitung der Daten, Evaluation des Fragebogens, Extraktion mit SubsMatch, Durchführung der verschiedenen Maschinellen Lernalgorithmen in Matlab, Durchführung von Multimatch und anschließender Präsentation dieser gewonnenen Daten.

5.1. Aufbereitung

Die erhobenen Daten befanden sich in Rohfassung in einem String. Jede Zeile in der Datei entspricht einem erhaltenen String an Daten. Das geschriebene Programm, im weiteren *ConvertTool* genannt, verarbeitet die Daten wie in folgendem Pseudo-Code beschrieben:

Algorithm 1: ConvertTool in Pseudo-Code

Data: Rohdaten vom Eye-Tracking
Result: extrahierte Daten aus den Rohdaten
Initialisierung;
while *noch ein Ordner in dem referenzierten Ordner* **do**
    while *noch eine Datei in dem Ordner* **do**
        Öffne Datei;
        while *nicht Ende der Datei* **do**
            Lade Zeile;
            Konvertiere Zeile in JSON-Objekt;
            Extrahiere Daten aus JSON-Objekt;
            Schreibe extrahierte Daten in neue Datei;
        end
        Schließe die Datei;
    end
end

```
while ((line = sr.ReadLine()) != null)
{
    // loading line
    Console.WriteLine(line);
    Console.WriteLine("\n");
    if (String.IsNullOrEmpty(line) || line == "Start of the eye tracking")
    {
        Console.WriteLine("String is Null or Empty (or useless) " + j);
        continue;
    }
    dynamic currentline = JObject.Parse(line);

    if ("tracker" != (string) currentline.category || 0 == Convert.ToDouble(currentline.values.frame.avg.x) ||
        0 == Convert.ToDouble(currentline.values.frame.avg.y))
    {
        Console.WriteLine("in here " + j);
        continue;
    }

    if (beginning && starttime == "")
    {
        starttime = (string) currentline.values.frame.timestamp;
        beginning = false;
    }
    // writing part
    // getting all variables set
    if (currentline.values.frame.avg.x != null || currentline.values.frame.avg.y != null)
    {
        xavg = currentline.values.frame.avg.x;
        yavg = currentline.values.frame.avg.y;
    }
    else
    {
        xavg = "0";
        yavg = "0";
    }
    currenttime = (string) currentline.values.frame.timestamp;
    timespan = TimeCalc(starttime, currenttime, j);
    wr.WriteLine(xavg + ";" + yavg + ";" + timespan);
    j += 1;
}
```

Abbildung 5.1.: Code des *While*-Loop im *ConvertTool*

Expliziter Code in Abbildung 5.1: Die erste Zeile der Datei wird entfernt, da es sich hierbei um eine Debug Zeile des Clienten aus dem Kapitel 4.1.3 handelt. Das Convert-Tool entfernt hierbei alle nicht benötigten Informationen aus den gemessenen Daten.

Dabei wird über jede Zeile einer Datei iteriert und die Informationen über den Zeitpunkt und die genauen Pixelkoordinaten extrahiert. Die so aufbereiteten Daten wurden für die Evaluation in Matlab genutzt.

5.2. Evaluation

Die Evaluation erfolgte in mehreren Schritten: Zuerst wurden für die Evaluation aufgenommene ungültige Daten, d.h. Aufnahmen, die viele Fehler oder aber eine sehr schlechte Kalibrierung aufwiesen, ausgenommen. Von 15 Probanden blieben nach dem Auswahlverfahren noch 12 Probanden übrig, welche für die Evaluation genutzt wurden. Von diesen 12 Probanden haben 8 Probanden sich für die Sprache Python entschieden. Daher eignet sich eine Untersuchung dieser Gruppe im Speziellen besonders. Jeder Proband, welcher mindestens eine Aufgabe zum Großteil (mehr als 50%) falsch hatte, wurde als Novize eingestuft. So entstanden unter allen Testpersonen 5 Novizen und 8 Experten.
Die Daten dieser 12 Probanden wurde dann mit dem unter 5.1. Aufbereitung genannten aus den Rohdaten extrahiert. Anschließend wurde der Fragebogen dieser 12 Probanden genauer betrachtet.

5.2.1. Fragebogen

Alle Fragebögen der gültigen 12 Probanden wurden ausgewertet. Es ergaben sich folgende Werte:

	Mittelwert	Varianz	Min.	Max.
Alter (in Jahren)	27,75	2,69	24	30
Zeit im Coding (in Jahren)	8,25	13,35	3	15
Eye-Tracking (in Jahren)	0,88	2,26	0	4
Selbsteinschätzung	7	2,5	3	10
Anzahl der verwendeten Monitore	2	0,33	1	3
Diagonale Größe in Zoll	23,44	11,27	15	27

Tabelle 5.1.: Auswertung des Fragebogens

Nach Auswertung der Fragebögen wurden mit den extrahierten Daten weitergearbeitet, um diese vergleichen zu können.

5.2.2. SubsMatch 2.0

Die extrahierten Daten werden in Matlab importiert und weiter verarbeitet: Eine Grid der Größe x wurde generiert. Anschließend wurden aus den vorbereiteten Daten mittels einer Fixations-Erkennungs-Funktion die Fixationen je Aufgabe und Person erkannt. Dadurch entstanden pro Person 8 Matrizen (je Aufgabe eine Matrix). Mit den Parametern von 500 Millisekunden für einen Buchstaben, einer gewünschten Subsequenzlänge von 3 und den Matrizen mit den Fixationen wurde die SubsMatch Funktion aufgerufen. Anhand dieser Daten wurden nun Subsequenzen (wie in 2.2.1 genauer erklärt) aufgebaut. So entstanden, je nach Größe des Grids, unterschiedlich große Matrizen welche die Informationen über Subsequenzen enthielten. Diese Matrizen wurden dann als Grundlage für den in Matlab integrierten Regression- bzw. Klassifikations-Learner genutzt. Die Klassifizierung bzw. Regression sollte im Idealfall zwischen Novizen und Experten unterscheiden können. Um die Ergebnisse möglichst detailliert und vergleichbar zu haben, wurden hierzu verschiedene Grids der Größe x genutzt ($x \in \{4,5,6,7,8,9\}$). Bei kleinen Grids wurde innerhalb von Matlab auf Cross-Validation (einzelne Teile der Datensätze werden zur Validierung gegenübergestellt) gesetzt. Bei den großen Grids aufgrund der Anzahl von Datenpunkten nur noch auf Holdout-Validation (ein Teil wird als Trainingsdaten genutzt und im Anschluss gegen einen anderen Teil des Gesamten validiert).

Da ein Großteil der Teilnehmer Python als Sprache ausgewählt hatte, wurde auch ein direkter Vergleich zwischen allen Python-Nutzern vorgenommen. Folgende unterschiedliche Auswertungen wurden vorgenommen:

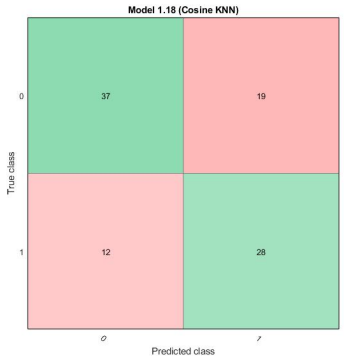
Klassifikation

- Probanden basiert: Probanden binär eingeteilt (alle Aufgaben)
- Probanden basiert: gemittelter String aller Aufgaben je Proband binär
- Probanden (Python) basiert: Probanden binär eingeteilt (alle Aufgaben)

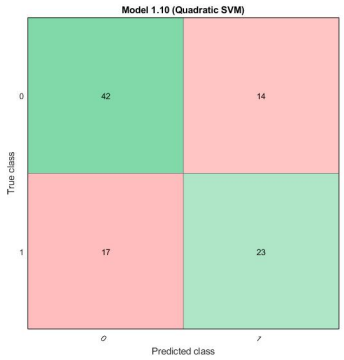
Die unterschiedlichen Grids x haben teilweise zu kaum Vielfalt geführt, daher werden hier nur die prägnantesten Ergebnisse aufgeführt.
Eine Auswertung über den Mittelwert der Aufgaben je Proband wurde durchgeführt, jedoch führte dies zu keinem brauchbaren Ergebnis. Die Werte lagen hierfür unter 50%.

Die in der Abbildung 5.2 gezeigten Konfusionsmatrizen zeigen eindeutig eine Unterscheidbarkeit, jedoch keine wirklich gute bzw. aussagekräftige. Der Wert liegt dabei zwischen 62,50% (5.2c) und 67,71% (5.2a).

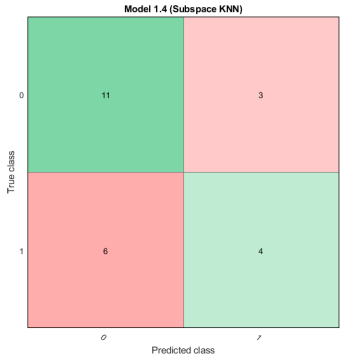
Abbildung 5.3 zeigt ähnlich zu Abbildung 5.2 einen Wert zwischen 64,06% (5.3b)



(a) Konfusionsmatrix Kosinus kNN (5x5 Grid)

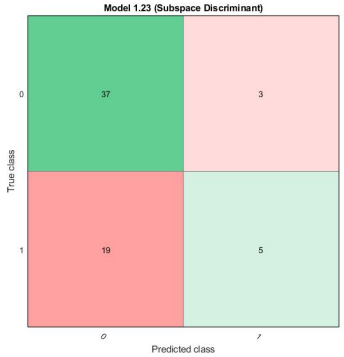


(b) Konfusionsmatrix Quadratische SVM (5x5 Grid)

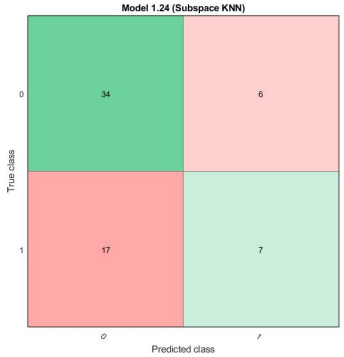


(c) Konfusionsmatrix Subspace kNN (8x8 Grid)

Abbildung 5.2.: Ergebnisse aller Nutzer in unterschiedlichen Grids



(a) Konfusionsmatrix Subspace Diskrimante



(b) Konfusionsmatrix Subspace kNN

Abbildung 5.3.: Ergebnisse der Python Nutzer in einem 5x5 Grid

und 65,63% (5.3a). Dies ist genau wie zuvor ein zu niedriger Wert für einen reellen Einsatz, deutet aber auf eine Unterscheidbarkeit hin.

Regression

Das Ziel der Regression ist, eine Korrelation zwischen den Daten zu finden. Sollte eine Korrelation bestehen, kann eine Vorhersage für das Expertise Level getroffen werden. Das Maschinelle Lernverfahren kann damit die Daten verallgemeinern und auf andere Fälle anwenden.

- Probanden basiert: gemittelter String aller Aufgaben je Proband binär
- Probanden (Python) basiert: Aufgaben Mittelwert binär eingeteilt
- Probanden (Python) basiert: Nutzer gemittelt (ein Nutzer erhält je falsche Aufgabe 0.125 Bonus)

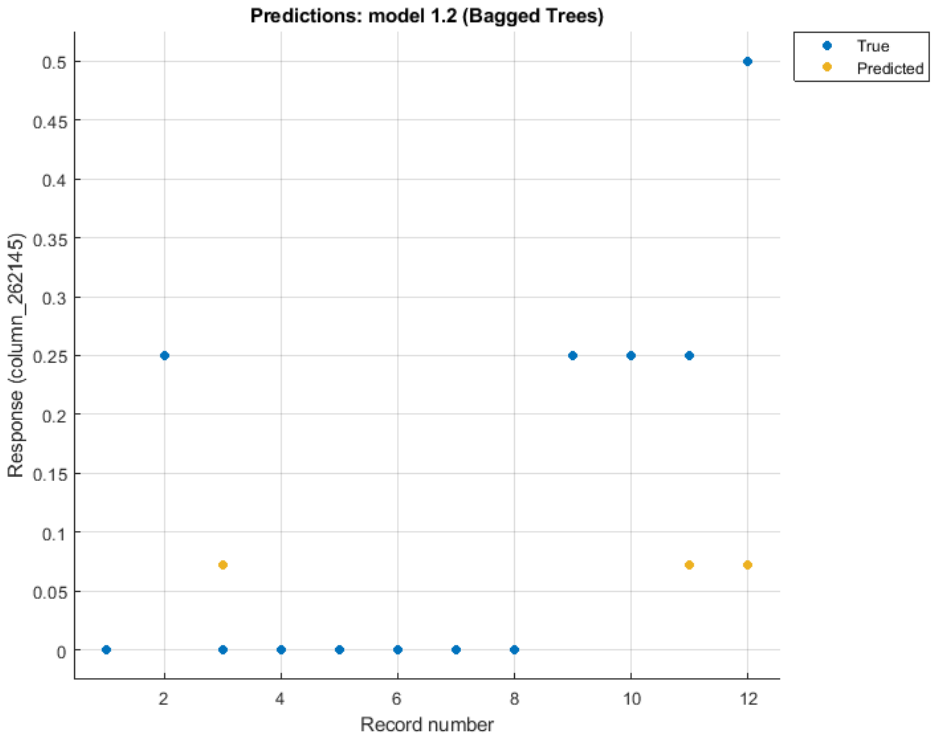
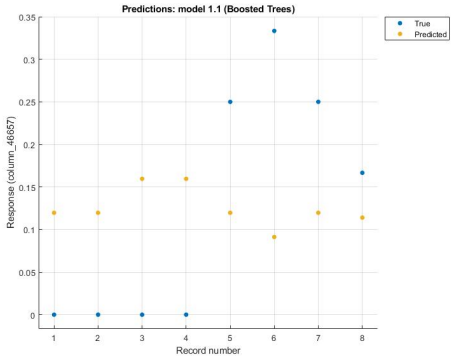


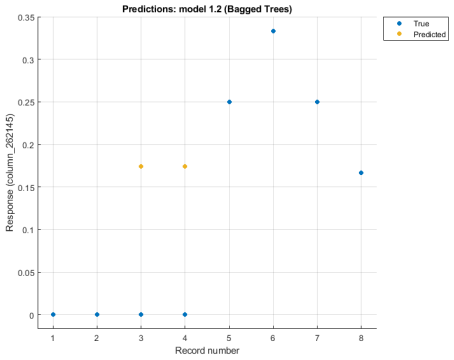
Abbildung 5.4.: Einschätzungsplot verschachtelter Entscheidungsbaum in einem 8x8 Grid je Nutzer gemittelter String aller Aufgaben

Der 8x8 verschachtelter Entscheidungsbaum ermöglicht eine relativ gute Regression, auch wenn bei dem letzten Punkt deutlich wird, dass die Regression immer den

selben Schwerpunkt setzt und nicht generalisiert wird.



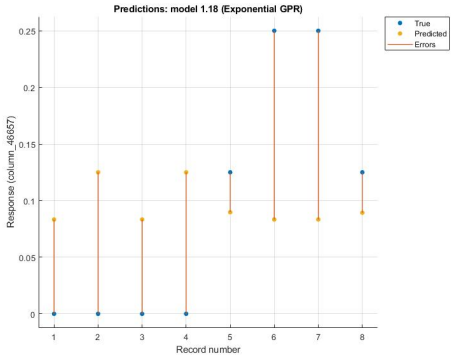
(a) Einschätzungsplot verstärkter Entscheidungsbaum in einem 6x6 Grid



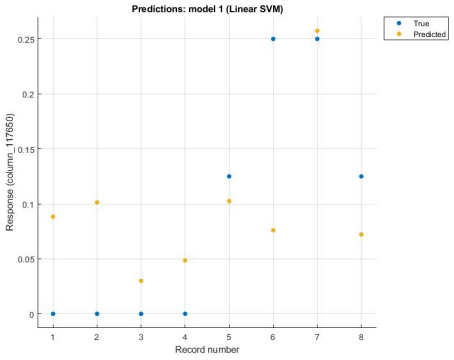
(b) Einschätzungsplot verschachtelter Entscheidungsbaum in einem 8x8 Grid

Abbildung 5.5.: gemittelter String je Aufgabe

Abbildung 5.5 zeigt mehr oder weniger über alle Punkte ein Raten. Kein Punkt wird wirklich richtig eingeschätzt.



(a) Einschätzungsplot Subspace Diskriminante 6x6 Grid



(b) Einschätzungsplot Subspace kNN 7x7 Grid

Abbildung 5.6.: Ergebnisse der Python Nutzer; je Nutzer einen gemittelten String der Aufgaben in einem 6x6 und einem 7x7 Grid

Die Regression auf nur Python-Nutzern (5.6) verlief im Vergleich zu den anderen Ansätzen am besten, jedoch auch hier ist in Abbildung 5.6a eine merkbare Abweichung vorhanden und Abbildung 5.6b wesentlich mehr Varianz in den einzelnen Punkten. Eine eindeutige Unterscheidbarkeit ist nicht möglich. Hier ist ein Einsatz in der reellen Welt ungeeignet.

Als Vergleich zu SubsMatch wurde Multimatch verwendet.

5.2.3. Multimatch

Für Multimatch wurde das Schema der Daten verändert, sodass diese die Form $X;Y;d$ besaßen, wobei d die Dauer der Fixation an der Position X,Y ist. Da der Input von Multimatch immer zwei einzelne Dateien sind, wurden alle 8 Aufgaben einer Person in einer Datei zusammengeführt. Diese Datei beinhaltet somit alle aufgenommenen Informationen einer Person. Anschließend wurden alle 12 einzelnen Dateien gegeneinander in Multimatch als Input verwendet. Multimatch bietet die Möglichkeit einer Vereinfachung des Scanpaths, indem z.B. ein größerer Unterschied im Winkel noch als gleich anerkannt wird. Es wurden keine Vereinfachungen an der Erkennung bzw. dem Vergleichen vorgenommen. Somit ergab sich eine gesamte Testanzahl von 66. Diese 66 Test-Ergebnisse beinhalten, aufgrund ihrer Aufstellung, auch alle Python Nutzer und können daher extrahiert werden. Die Ergebnisse wurden auf 4 Nachkommastellen gerundet.

Anschließend wurden die besten Repräsentanten für die Gruppe Novize bzw. Experte herausgesucht. Diese wurden dann gegen die übrigen Teilnehmer in Multimatch gestellt. Als Schranke für die Klassifizierung wurde $s = 0{,}008$ angesetzt. Die entstanden Konfusionsmatrizen sind die folgenden:

		True	
		1	0
Predicted:	1	4	1
	0	2	4

(a) Konfusionsmatrix aller Nutzer mit dem Repräsentanten für Experten. Klasse 1 ist hierbei die Expertenklasse.

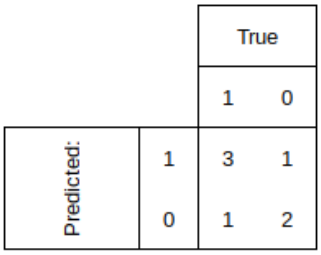
		True	
		1	0
Predicted:	1	3	2
	0	1	5

(b) Konfusionsmatrix aller Nutzer mit dem Repräsentanten für Novizen. Klasse 1 ist hierbei die Novizenklasse.

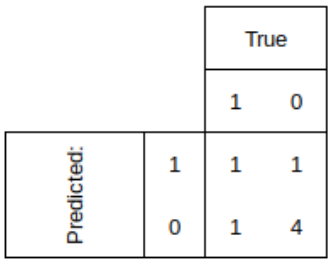
Abbildung 5.7.: Konfusionsmatrizen aller Nutzer

Multimatch gelingt es in Abbildung 5.7a 72,7% mit dem Experten als Vergleich richtig einzuteilen. In Abbildung 5.7b ebenso zu 72,7% mit dem Repräsentanten der Novizen.

Auch innerhalb der Python-Nutzer (Abbildung 5.8) ist die Unterteilung in Novize und Experte nicht eindeutig. So beträgt in Abbildung 5.8a die Genauigkeit 71,4%. Ebenso in Abbildung 5.8b.



(a) Konfusionsmatrix der Pythonnutzer mit dem Repräsentanten für Experten. Klasse 1 ist hierbei die Expertenklasse.



(b) Konfusionsmatrix der Pythonnutzer mit dem Repräsentanten für Novizen. Klasse 1 ist hierbei die Novizenklasse.

Abbildung 5.8.: gemittelter String je Aufgabe

Die Statistik (Mittelwert) von Experten und Novizen ist in Tabelle 5.2 dargestellt. Es fällt auf, dass der Unterschied zwischen den zwei Klassen von Experten und Novizen nicht sehr groß ist.

Experten:	Vektor	Richtung	Länge	Position	Dauer
Mittelwert:	0,9577	0,7589	0,9514	0,8688	0,2548
Varianz:	0,0000	0,0007	0,0000	0,0006	0,0011
Novizen:	Vektor	Richtung	Länge	Position	Dauer
Mittelwert:	0,9507	0,7839	0,9423	0,8800	0,2926
Varianz:	0,0000	0,0007	0,0001	0,0003	0,0021
Unterschied:	0,0070	0,0250	0,0091	0,0113	0,0378

Tabelle 5.2.: Der Mittelwert von Novizen und Experten mit dem absoluten Unterschied zwischen diesen in der letzten Zeile

6. Diskussion

Wenn ein Unterschied zwischen Novizen und Experten in der Betrachtung von Code besteht, sollte dieser nach einer Merkmalsextraktion auch von Maschinellem Lernen erkannt werden. Um zu bewerten ob dies möglich ist, wurden die Daten wie folgt analysiert: Zuerst wurde der Fokus auf die Ergebnisse des Fragebogens gelegt. Danach wurde der Merkmalsextraktionsalgorithmus SubsMatch aufgeteilt in die Auswertung von Klassifizierungs- und Regressions-Learner betrachtet und die ermittelten Werte interpretiert. Multimatch wurde als Vergleich hierzu angwendet. Abschließend wurden die verschiedenen Ergebnisse verglichen.

6.1. Fragebogen

Der Fragebogen wurde so konzipiert, dass von den Probanden eine Selbsteinschätzung generiert wurde. Außerdem wurden die Punkte Monitor und Größe zur Ermittlung der Vergleichbarkeit zum aktuellen Aufbau aufgenommen. Würden die Probanden beispielsweise alle weit mehr als 2 Bildschirme nutzen oder aber kleinere oder größere, könnte dies einen Einfluss auf ihre Fähigkeiten haben. Dies war jedoch nicht der Fall.
Des Weiteren weist der Fragebogen in Tabelle 5.1 mit einem Mittelwert von 8 Jahren und 3 Monaten Coding-Erfahrung einen relativ hohen Wert und somit viel Erfahrung auf. Unter Hinzunahme von dem Durchschnittsalter von 27 Jahren und 9 Monaten ergab sich so im Schnitt eine Beschäftigung mit Programmieren von 29,75% anteilig am gesamten Leben der Probanden. Der Durchschnitt der Selbsteinschätzung lag bei 7 von 10. Als Schlussfolgerung aus dem Fragebogen waren die teilnehmenden Probanden verhältnismäßig gute Programmierer. Außerdem war der Versuchsaufbau nah an deren Produktivumgebung orientiert.

6.2. Eye-Tracking Daten

Die aufgenommenen Daten der Probanden wurden wie folgt binär dargestellt. Die Aufteilung erfolgte in 0 und 1, wobei die 0 ein Experten und die 1 einen Novizen mit Fehlern in den Aufgaben darstellte.

Aufgabe	Verständnis				Fehler			
Sprache	1	2	3	4	1	2	3	4
Python	0	0	0	0	0	0	0	0
Python	0	0	0	0	1	0	1	0
Java	0	0	0	0	0	0	0	0
C++	0	0	0	0	0	0	0	0
Python	0	0	0	0	0	0	0	0
Python	0	0	0	0	0	0	0	0
Python	0	0	0	0	0	0	0	0
Python	0	0	0	0	0	0	0	0
C++	0	0	0	0	1	1	0	0
Python	0	0	0	0	0	1	0	1
Python	0	0	0	0	0	1	1	0
C++	0	0	0	0	1	1	1	1

Tabelle 6.1.: Auswertung der Eye-Tracking Daten je Aufgabe - jeder Proband mit einem Fehler gilt als Novize (grau hinterlegt)

6.3. SubsMatch

6.3.1. Klassifizierung

Eine Auswertung der Maschinellen Lernalgorithmen zum Klassifizieren der Abbildungen aus dem vorherigen Kapitel sah dabei wie folgt aus:

Python	
Subspace Diskriminante	65,63 %
Subspace KNN	64,06 %

Tabelle 6.2.: Die Auswertung der Konfusionsmatrix der Python-Probanden gemittelt über alle Grid-Größen und auf 2 Nachkommastellen gerundet

und

User	
Subsace KNN	62,50 %
Coarse Tree	62,50 %
Cosine Tree	67,71 %
Quadratic SVM	67,71 %

Tabelle 6.3.: Die Auswertung der Konfusionsmatrix aller Probanden gemittelt über alle Grid-Größen und auf 2 Nachkommastellen gerundet

Die Werte liegen alle etwas über 60% und keiner erreicht die 70%. Dies war bei allen Tests in Matlab mit den aufgelisteten Algorithmen gleich. Einige erreichten gerade so die 50%. Dieser Wert von 50% entspricht dem puren Erraten der Klasse. Daher entsteht die Annahme, dass diese Algorithmen in einer Klassifizierung keine klare Einteilung in Novize und Experte vornehmen können.

6.3.2. Regression

Das Ziel der Regression ist es, zwischen den Daten eine Korrelation herzustellen und diese zu verallgemeinern. Im Idealfall kann diese Korrelation zum Klassifizieren genutzt werden.
Die Regression ist, wie in den Abbildungen aus dem vorherigen Kapitel zu entnehmen, nicht allzu gut verlaufen. Die Werte liegen in allen Fällen nicht oder nicht sehr nahe an den gemessenen Werten. In Abbildung 5.5b wurde mit dem 8x8 Grid eine gute Näherung bestimmt, jedoch war eine genaue Bestimmung nicht möglich. Dies verstärkt die Annahme, dass eine Einteilung in Novize und Experte nicht ohne weiteres möglich ist.

6.4. Multimatch

Die Klassifikation mit Multimatch verlief im Vergleich zu Subsmatch erfolgreicher. Ein Wert von über 70% (z.B. Abbildung 5.7a mit 72,7%) wurde in allen Klassifikationen erreicht. Jedoch ist dieser Wert nicht allzu aussagekräftig, da die Anzahl der Samples gering ist. Dennoch zeigt sich auch hier eine Unterscheidbarkeit in Experte und Novize.

6.5. Vergleich

Die extrahierten Merkmale waren vermutlich nicht optimal oder es gab keinen großen Unterschied innerhalb der getesten Personen, sodass die Maschinellen Lernalgorithmen keine gute Unterscheidung aufgrund der extrahierten Merkmale vornehmen konnten. Unter Hinzunahme des Fragebogens wird klarer, dass die getesteten Probanden sowohl in der Selbsteinschätzung als auch in Erfahrung in Jahren einen enorm hohen Wert aufweisen. Daher ist dieses Ergebnis mit den vorliegenden Probanden nicht sehr überraschend und bedarf weiterer Untersuchung mit einer größeren Vielfalt und Anzahl an Probanden.

7. Fazit

Die im vorangegangenen Kapitel aufgeführten Daten weisen keine guten Unterscheidungsmerkmale von SubsMatch mit einer Grid basierten Extraktion auf, sodass die Maschinellen Lernalgorithmen nicht gut mit den aufgenommenen Daten umgehen können und eine Einteilung schwer fällt. Ein mittlere Genauigkeit von 65,02% über alle ausgeführten Maschinellen Lernalgorithmen auf den Daten von SubsMatch ist kein guter Wert für eine Klassifikation in Anbetracht der Tatsache, dass es eine binäre Einteilung der Probanden ist. Bei 50% wäre es ein reines Raten. Bei MultiMatch ist die mittlere Genauigkeit mit 72,08% höher, aber ermöglicht auch keine gute Einteilbarkeit.
Ein weiteres Indiz für keine gute Einteilung der Probanden in Novize und Experten ist der Fragebogen, bei welchem sich alle Probanden nicht nur selbst gut einschätzen, sondern auch eine lange Erfahrung von durchschnittlich 8 Jahren und 3 Monaten als Programmierer aufweisen. Auch eine Auswertung ausschließlich der Python Nutzer brachte keine entsprechenden Ergebnisse. Die genutzten Probanden waren vermutlich alle zu erfahren, um eine klare Unterscheidung in Novize und Experten zu ermöglichen. Multimatch unterstreicht dies mit der Statistik in Tabelle 5.2 noch weiter mit höheren Werten der Ähnlichkeit.

Im Vergleich zu den unter Kapitel 3 erwähnten Arbeiten ist auch ein Unterschied zu erkennen: Die neuesten Publikationen nutzen als Vergleichsgruppe erfahrene Programmierer und angehende Programmierer mit sehr wenig Erfahrung. Eine Publikation [UNMiM06] weist einen Unterschied in der Gesamtdauer der Betrachtung und der Fixationspunkte hin, jedoch konnte dies in dieser Studie nicht bestätigt werden.
Die Auswahl der Teilnehmer wird der entscheidende Punkt gewesen sein. In den anderen Publikationen weisen die Teilnehmer wesentlich mehr Varianz in der Erfahrung auf.

8. Ausblick

Das Fazit der Studie fiel ernüchternd aus und eine Einteilung in Novize und Experte ist aufgrund von fehlenden eindeutigen Merkmalen der Probanden und einer vermutlich zu kleinen Testgruppe nicht möglich. Ein Problem dabei ist wahrscheinlich auch die fehlende Vielfalt in der Erfahrung der Probanden. Jedoch kann in diese Richtung weiter geforscht werden unter dem Aspekt, dass im Voraus eindeutig unterschiedliche Erfahrungslevel vorliegen und explizit nach diesen gesucht wird. In weiterführenden Experimenten muss der Unterschied in der Expertise der Probanden größer sein und die Auswahl dieser genauer vorgenommen werden. Außerdem ist eine größere Testgruppe zu empfehlen.

A. Literaturverzeichnis

[BDL+12] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2):219–276, may 2012.

[Bla] A.F. Blackwell. First steps in programming: a rationale for attention investment models. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*. IEEE Comput. Soc.

[BS20] Brillen-Sehhilfe. Website. *https://www.brillen-sehhilfen.de/auge/*, 2020. Abgerufen am 30.03.2020.

[BSB11] Teresa Busjahn, Carsten Schulte, and Andreas Busjahn. Analysis of code reading to gain more insight in program comprehension. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research - Koli Calling '11*. ACM Press, 2011.

[BT06] Roman Bednarik and Markku Tukiainen. An eye-tracking methodology for characterizing program comprehension processes. In *Proceedings of the 2006 symposium on Eye tracking research & applications - ETRA '06*. ACM Press, 2006.

[Bus22] G.T. Buswell. *Fundamental reading habits: a study of their development*. Monog.: Chicago, 1922.

[Car03] John M. Carroll. Introduction: Toward a multidisciplinary science of human-computer interaction. In *HCI Models, Theories, and Frameworks*, pages 1–9. Elsevier, 2003.

[CAS17] K R Chandrika, J Amudha, and Sithu D Sudarsan. Recognizing eye tracking traits for source code review. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, sep 2017.

[CSW02] Martha E. Crosby, Jean Scholtz, and Susan Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In *Programmers, 14th Workshop of the Psychology of Programming Interest Group, Brunel University*, pages 18–21, 2002.

[DNJ+12] Richard Dewhurst, Marcus Nyström, Halszka Jarodzka, Tom Foulsham, Roger Johansson, and Kenneth Holmqvist. It depends on how you look at it: Scanpath comparison in multiple dimensions with MultiMatch, a vector-based approach. *Behavior Research Methods*, 44(4):1079–1100, may 2012.

[Eye16a] EyeTribe. Website. *https://theeyetribe.com/theeyetribe.com/about/index.html*, 2016. Abgerufen am 08.02.2020.

[Eye16b] EyeTribe. Website-dev. *gekürzt als https://cutt.ly/ztwHVTL*, 2016. Abgerufen am 08.02.2020.

[Fra17] John M. Franchak. Head-mounted eye tracking. In *The Cambridge Encyclopedia of Child Development*, pages 113–116. Cambridge University Press, 2017.

[GJG04] Laura A. Granka, Thorsten Joachims, and Geri Gay. Eye-tracking analysis of user behavior in WWW search. In *Proceedings of the 27th annual international conference on Research and development in information retrieval - SIGIR '04*. ACM Press, 2004.

[HN12] Prateek Hejmady and N. Hari Narayanan. Visual attention patterns during program debugging with an IDE. In *Proceedings of the Symposium on Eye Tracking Research and Applications - ETRA '12*. ACM Press, 2012.

[Hue08] Edmund Burke Huey. *The Psychology and Pedagogy of Reading*. New York: Macmillan, 1908.

[KKR14] Thomas C. Kübler, Enkelejda Kasneci, and Wolfgang Rosenstiel. SubsMatch. In *Proceedings of the Symposium on Eye Tracking Research and Applications - ETRA '14*. ACM Press, 2014.

[KRJ17] Chandrika K R and Amudha Joseph. An eye tracking study to understand the visual perception behavior while source code comprehension. 01 2017.

[KRS+16] Thomas C. Kübler, Colleen Rothe, Ulrich Schiefer, Wolfgang Rosenstiel, and Enkelejda Kasneci. SubsMatch 2.0: Scanpath comparison and classification based on subsequence frequencies. *Behavior Research Methods*, 49(3):1048–1064, jul 2016.

[KWK+13] Rami N. Khushaba, Chelsea Wise, Sarath Kodagoda, Jordan Louviere, Barbara E. Kahn, and Claudia Townsend. Consumer neuroscience: Assessing the brain response to marketing stimuli using electroencephalogram (EEG) and eye tracking. *Expert Systems with Applications*, 40(9):3803–3812, jul 2013.

[Mat20a] Matlab. Website. *https://www.mathworks.com/products/matlab.html*, 2020. Abgerufen am 23.02.2020.

[Mat20b] Matlab. Website. *https://www.mathworks.com/help/stats/decision-trees.html*, 2020. Abgerufen am 23.02.2020.

[Mat20c] Matlab. Website. *https://www.mathworks.com/help/stats/discriminant-analysis.html*, 2020. Abgerufen am 23.02.2020.

[Mat20d] Matlab. Website. *https://www.mathworks.com/help/stats/classification-naive-bayes.html*, 2020. Abgerufen am 23.02.2020.

[Mat20e] Matlab. Website. *https://www.mathworks.com/help/stats/support-vector-machines-for-binary-classification.html*, 2020. Abgerufen am 23.02.2020.

[Mat20f] Matlab. Website. *https://www.mathworks.com/help/stats/classificationknn.html*, 2020. Abgerufen am 23.02.2020.

[Mat20g] Matlab. Website. *https://www.mathworks.com/help/stats/classification-ensembles.html*, 2020. Abgerufen am 23.02.2020.

[Not19] Notepad++. Website. *https://notepad-plus-plus.org/*, 2019. Abgerufen am 13.02.2020.

[Res20] SR Research. Website. *https://www.sr-research.com/eyelink-1000-plus/*, 2020. Abgerufen am 06.04.2020.

[SAB+18] Lauren K. Slone, Drew H. Abney, Jeremy I. Borjon, Chi hsin Chen, John M. Franchak, Daniel Pearcy, Catalina Suarez-Rivera, Tian Linger Xu, Yayun Zhang, Linda B. Smith, and Chen Yu. Gaze in action: Head-mounted eye tracking of children's dynamic visual attention during naturalistic behavior. *Journal of Visualized Experiments*, (141), nov 2018.

[SFM12] Bonita Sharif, Michael Falcone, and Jonathan I. Maletic. An eye-tracking study on the role of scan time in finding source code defects. In *Proceedings of the Symposium on Eye Tracking Research and Applications - ETRA '12*. ACM Press, 2012.

[TFSL14] Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. An eye-tracking study assessing the comprehension of c++ and python source code. In *Proceedings of the Symposium on Eye Tracking Research and Applications - ETRA '14*. ACM Press, 2014.

[UNMiM06] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken ichi Matsumoto. Analyzing individual performance of source code review using reviewers' eye movement. In *Proceedings of the 2006 symposium on Eye tracking research & applications - ETRA '06*. ACM Press, 2006.

[Web20] WebMD. Website. *https://www.webmd.com/eye-health/amazing-human-eye*, 2020. Abgerufen am 05.04.2020.

[Wik20] Wikipedia. Website. *gekürzt als https://cutt.ly/HtwHL3T*, 2020. Abgerufen am 03.03.2020.

[Yar67] A. L. Yarbus. *Eye Movements and Vision*. New York: Plenum Press, 1967.

B. Abbildungsverzeichnis

C. Tabellenverzeichnis