HOCHSCHULE COBURG

Hochschule für angewandte Wissenschaften Coburg

Fakultät Elektrotechnik und Informatik

Studiengang: Informatik

Projekt Robotik - Machine Learning basierte Authentifizierung mit Hilfe einer Microsoft Kinect

Daniel Kirchner

Hinweis

Sie sollten zusammen mit dieser Arbeit die folgenden Dateien erhalten haben:

- **load.py** - Dieses Python-Skript enthält die Implementierung der Datenverarbeitung aus Kapitel 2.3
- **svm.py** - Dieses Python-Skript enthält die Implementierung der Support Vector Machine aus Kapitel 2.4
- **tree.py** - Dieses Python-Skript enthält die Implementierung des Entscheidungsbaumes aus Kapitel 2.5
- **raw_data/** - Dieser Ordner enthält einen Ausschnitt der Trainingsdaten für die Klassifizierungsalgorithmen

Falls dem nicht so ist sind diese Dateien auch in einem git-Repository unter
`https://github.com/DanielKirchner/ropr-ml/`
verfügbar.

Inhaltsverzeichnis

Abbildungsverzeichnis

Codebeispielverzeichnis

Tabellenverzeichnis

1 Robotik Projekt

Das *Robotik Projekt* ist ein für technische Studiengänge angebotenes Wahlpflichtfach, bei dem Studenten aus einer gegebenen Liste ein Projekt zur Bearbeitung für ein Semester auswählen können. In meinem Fall kommt das Projekt von einem Masterstudenten aus dem Studiengang Elektrotechnik und befasst sich mit der Authentifizierung mit Hilfe einer Microsoft Kinect.

1.1 Projektbeschreibung

Der von mir bearbeitete Teilbereich des Projekts beschränkt sich auf das softwareseitige Auswerten der Rohdaten.
Geliefert werden diese von einer Kinect, was eine Hardware der Firma Microsoft ist. Diese ermöglicht es, mit Hilfe eines Tiefensensor und einer Kamera räumliche Vorgänge aufzunehmen.[1] Auch enthält die Kinect-API[1] ein System zur Erkennung von Personen und liefert diverse Information (z.B. Position im Raum, Haltung) über diese zurück.
Das Ziel des Projektes ist es, zu bewerten, welcher Machine Learning Ansatz für das eindeutige Klassifizieren von Probanden anhand der Daten der Kinect geeignet ist. Sollte dies funktionieren, kann der Ansatz zum Beispiel als Authentifizierungsmethode verwendet werden. In Frage kommen hierfür sowohl ”konventionelle” Algorithmen (SVM,Naive Bayes,etc.) als auch neuronale Netze. Aufgrund meiner Erfahrung im Umgang mit Support Vector Machines, wurden diese als Startpunkt zur Recherche gewählt.

1.2 Zeitplan

Ein wichtiger Aspekt der Projektplanung ist der Zeitplan, welcher sich in diesem Fall wie folgt gestaltete:

[1] s. https://docs.microsoft.com/en-us/previous-versions/windows/kinect/dn758675(v%3dieb.10)

Arbeitspaket	Beschreibung	geplanter Zeitraum
Einarbeitung SVM	Einarbeitung in die Thematik *Support Vector Machines*	2 Woche
Finden von Alternativlösungen	Finden von Alternativlösungen im Machine Learning Bereich (zu SVMs)	3 Wochen
Planen der Softwarestruktur	In diesem Arbeitspaket soll vor allem herausgefunden werden, ob eine eigene Implementierung der Machine Learning Algorithmen sinnvoll ist. Alternativ sollen adäquate Frameworks gefunden werden.	1 Wochen
Implementierung der Algorithmen	Implementieren der ausgewählten Algorithmen (im gewählten Framework).	3 Wochen
Auswertung	Testen und Auswerten der Programme. Festlegen und Anwenden von Unterscheidungskriterien. Schreiben dieser Arbeit.	3 Wochen

2 Projektumsetzung

Dieses Kapitel beschäftigt sich mit der Umsetzung des in *Projektbeschreibung* beschriebenen Projekts. Zunächst wird erläutert, welche Frameworks und Programmiersprachen zur Umsetzung verwendet wurden. Danach soll erklärt werden, wie die benötigten Daten beschafft wurden, in den folgenden Unterkapiteln werden dann einzelne Algorithmen und deren Besonderheiten vorgestellt werden.

2.1 Softwarestruktur

Ob eine eigene Implementierung der Algorithmen sinnvoll ist war eine wichtige Entscheidung im Rahmen dieses Projekts. Letztendlich war jedoch der zeitliche Aufwand dafür zu groß, der Zeitraum für das Projekt wäre wahrscheinlich nicht einhaltbar gewesen.
Dementsprechend musste auf Implementierungen in fertigen Frameworks zurückgegriffen werden. Die beiden attraktivsten Optionen waren hier *scikit-learn*[1] und *mlr* [2].
Letzteres ist ein im professionellen Bereich oft verwendetes Framework, welches jedoch die Verwendung der Programmiersprache *R* voraussetzt. Da ich diese leider nicht beherrsche musste weiter nach einer Alternative gesucht werden.
scikit-learn ist ein in *Python* implementiertes Framework mit sehr aktiven Entwicklern und einer großen Auswahl an diversen Machine Learning Algorithmen. Es ist unter der BSD Lizenz frei (sogar für Unternehmen) verfügbar.
Da ich in bereits Erfahrung im Umgang mit *Python* Programmierung habe entschied ich mich auch letztendlich für die Verwendung von *scikit-learn*.

2.2 Datenbeschaffung

Alle Machine Learning Algorithmen benötigen einen Trainingsdatensatz. Um solide Klassifizierungen zu erreichen, müssen genügend Trainingsdaten geliefert werden. Da die Anzahl der benötigten Daten je nach Algorithmus variiert, werden einfach so viele Daten wie möglich von so vielen Personen wie möglich erfasst.

Im Rahmen dieses Projektes kommen die Trainingsdaten von einigen Freiwilligen, welche sich in verschiedenen Positionen vor der Kinect aufstellten. Der Ablauf hierfür war für alle Freiwilligen:

1. In kurzem Abstand mit dem Gesicht zu der Kinect stehen

[1] Homepage: `http://scikit-learn.org/stable/index.html`
[2] Repository: `https://github.com/mlr-org/mlr`

2. Um 180°auf der Stelle drehen
3. Einige Meter geradeaus laufen
4. In größerem Abstand zur Kinect stehen bleiben
5. Um 180°auf der Stelle drehen

Zwischen den einzelnen Ablaufschritten finden jeweils kurze Pausen statt, da die Kinect ca. 30 Aufnahmen pro Sekunde nimmt und so noch mehr Trainingsdaten aufgenommen werden können.
Der durch dieses Verfahren entstehende Datensatz wurde von 11 Personen erfasst, was in 210160 einzelnen Datenpunkten[3] resultierte. s

2.3 Datenbearbeitung

Die Rohdaten liegen im folgenden Format vor:

```
...
FootLeft;-0,03828041;-0,5859481;1,460664;Tracked;
HipRight;0,1121507;0,1015209;1,700809;Tracked;
AnkleRight;1513719;-0,7247724;1,501014;Inferred;
...
```

Code 1: Rohdaten der Kinect

Es handelt sich hierbei um eine .csv-Datei, in welcher die einzelnen Spalten durch Semikolon separiert sind. Der Eintrag der ersten Spalte enthält den Namen des erkannten Körperteils, während in der zweiten und dritten Spalte dessen Koordinaten eingetragen sind.
Die letzte Spalte enthält einen Status über die Messung des Körperteils. Diese kann die beiden Zustände *Inferred* und *Tracked* annehmen, wobei der Normalfall hierbei *Tracked* ist. In den Zeilen, deren Status *Inferred* ist, wurde die Position eines nicht durch die Hardware erkannten Körperteils errechnet [2]. Datenpunkte, die errechnete Körperteilpositionen enthalten, werden aussortiert.

In der folgenden Abbildung ist eine visuelle Darstellung eines einzelnen Datenpunktes gezeigt.

[3] Ein Datenpunkt enthält alle 25 möglichen erkannten Körperteile

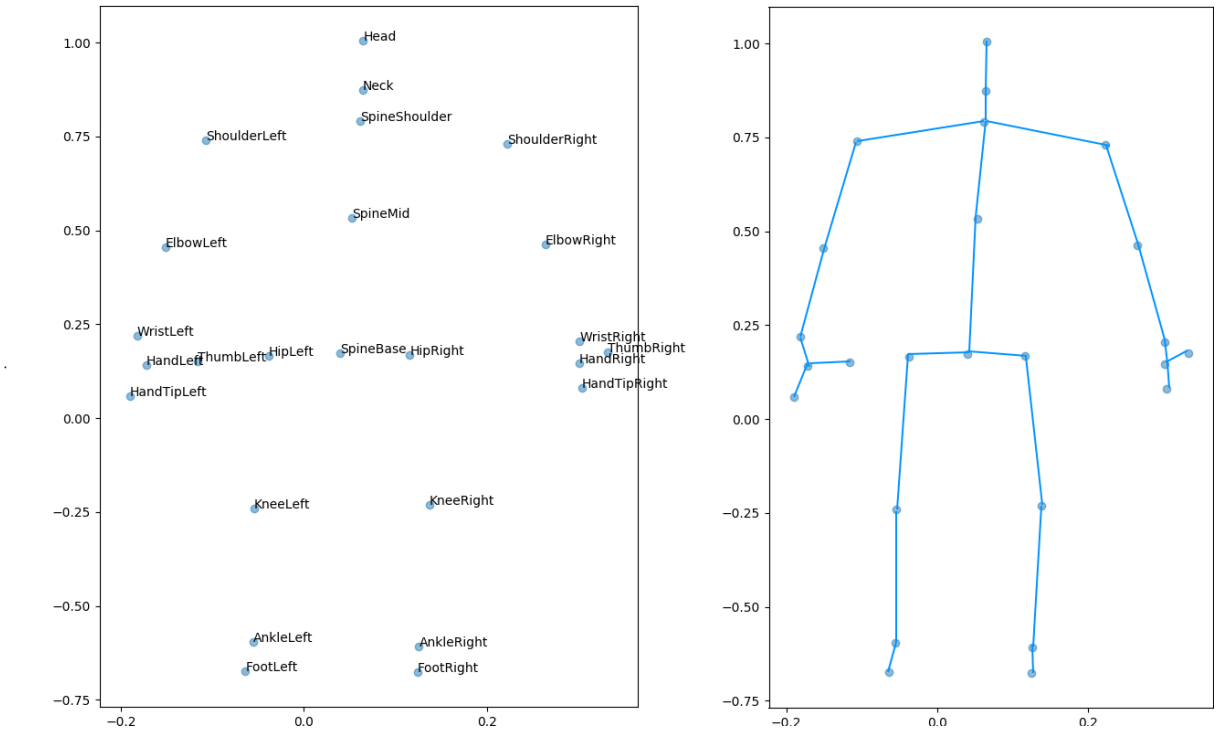

Abbildung 1: Visualisierung von Kinect-Rohdaten, links: Körperteile und deren Bezeichnungen, rechts: Koordinaten der Körperteile mit eingezeichnetem Skelet, gezeichnet mit matplotlib

Um weniger Features (Inputs für die Algorithmen) zu erreichen, werden die Daten reduziert. Das heißt in diesem Fall, dass mehrere Körperteilkoordinaten in die Länge des Körperteils zusammengefasst werden. Hierbei geht allerdings auch Information über z.B. die Winkel zwischen den Körperteilen verloren. Sollte also mit diesen reduzierten Inputdaten keine hohe Präzision in der Klassifizierung erreichbar sein, könnten die Winkel und andere Information als zusätzliche Features wieder später eingefügt werden.
Eine reduzierte Inputdatei sieht nun so aus:

```
...
neck;0.0835306;
shoulderHalfRight;0.177566;
upperarmRight;0.27685;
...
```

Code 2: Reduzierte Daten

Insgesamt wird die Inputinformation so von 25 Körperteilkoordinaten auf die 16 Körperteillängen reduziert.
Um die Genauigkeit eines Algorithmus zu ermitteln, sollten dafür Datenpunkte verwendet

werden, die nicht in den Trainingsdaten enthalten waren. So sieht der Algorithmus diese Daten ”zum ersten Mal” und kann keine primitive Abbildung von Inputwerten auf Outputwerte bilden. Falls zwar eine hohe Genauigkeit auf den Trainingsdaten erzielt werden kann, jedoch nicht mit bisher unbekannten Werten spricht man von ”Overfitting”.
Um diesen Effekt zu vermeiden werden die vorhandenen Daten in zwei Gruppen unterteilt. Die erste Gruppe wird dem Algorithmus zum trainieren gegeben, während die zweite Gruppe nur zu dessen Evaluation verwendet wird. Ich habe mich für eine 80-20 Aufteilung zwischen diesen Gruppen entschieden, wobei die Trainingsdaten die größere Gruppe sind.
Die gesamte in diesem Abschnitt beschriebene Vorgang ist in der Methode *load()* (Datei: **load.py**) implementiert.

2.4 Support Vector Machine

Im Jahre 1963 veröffentlichten Vapnik und Lerner ihre Formulierung des *Generalized Portrait Algorithm*, welcher ein linearer Klassifizierungsalgorithmus ist [3]. Die allgemeinere, nicht lineare Formulierung dieses Algorithmus wurde in [4] vorgestellt und ähnelt der heutigen Definition einer SVM.

2.4.1 Funktionsweise

Eine Support Vector Machine unterteilt Punkte zweier Klassen durch eine Gerade und bestimmt dann die Klasse von neuen Punkten daran, wie diese relativ zur Gerade liegen.
Die gegebenen Trainingsdaten werden zur Maximierung des Abstandes dieser Unterteilungsgerade zu den ihr am nächsten liegenden Punkten der beiden Klassen verwendet. Anhand der folgenden Abbildung soll die Verwendung der Support Vector Machine verdeutlicht werden.

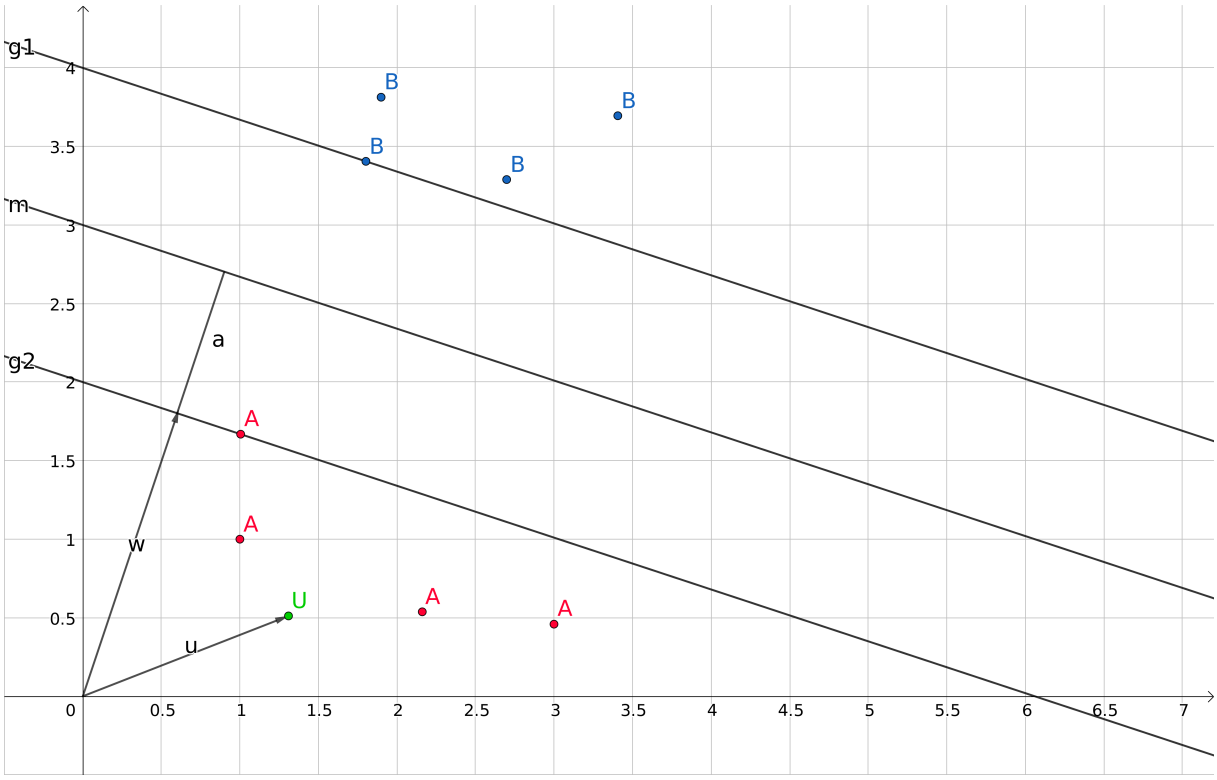

Abbildung 2: Funktionsweise der Support Vector Machine

Die rot dargestellten Punkte sind hier Mitglieder der Klasse A, während die blauen Punkte Mitglieder der Klasse B sind. Die Gerade m teilt die beiden Klassen voneinander.
Die Geraden $g1$ und $g2$ sind parallel zu m und haben jeweils den gleichen Abstand zu m. Diese sind die sogenannten Stützvektoren (eng. *support vectors*). Um eine optimale Klassifizierung gewährleisten zu können wird nun die Gerade m gesucht, für die der Abstand zu den nächsten Punkten beider Klassen a maximal ist.
Der Vektor w zeigt rechtwinklig auf m und ist von beliebiger Länge.
Um nun festzustellen, in welcher Klasse ein neuer, nicht klassifizierter, Punkt U liegt, wird dessen Ortsvektor u auf w projiziert und überprüft ob dieser größer als eine Konstante c ist:

$$w \cdot u \geq c$$

Die Konstante c ist hierbei der Abstand von m zum Ursprung. In diesem Beispiel also:

$$\left|w\right|+a$$

Trifft die oben genannte Bedingung zu, so wird U als Punkt der Klasse B klassifiziert, ansonsten als Punkt aus A. Die x-Komponente aller Punkte könnte in diesem Beispiel etwa die Länge des Beines der Testperson sein, während die y-Komponente die Länge des Nackens darstellt.
Support Vector Maschinen funktionieren jedoch nicht nur im zweidimensionalen Raum, sondern

in jedem beliebig-dimensionalen Raum. Die Trennung im dreidimensionalen Raum würde durch eine Fläche geschehen.
Da in unserem Fall ein Problem mit mehr als zwei Klassen (11 Personen) vorliegt, werden mehrere verschiedene SVMs trainiert, wobei diese jeweils ein *one-vs-rest*-Problem lösen. Dieses hat wiederum nur zwei Klassen (Klasse, nicht-Klasse) und ist darum für eine SVM geeignet.
Weitere Eigenschaften von SVMs (z.B. Umgang mit nicht linear lösbaren Problemen) werden in dieser Arbeit nicht behandelt, können jedoch in [4] nachgelesen werden.

2.4.2 Umsetzung

sklearn bietet zwei verschiedene Klassifizierungsalgorithmen für lineare SVMs an: *svm.SVC* und *svm.LinearSVC*. *svm.SVC* verwendet die C++ Bibliothek *libsvm*, welche verschiedene Kernelfunktionen (andere als linear) zulässt, während die C Bibliothek *LinearSVC* zwar nur die lineare Kernelfunktion implementiert, jedoch effizienter programmiert ist.
Da Performanz beim Training im Rahmen dieses Projektes nicht als Priorität gesehen wird und andere Kernelfunktionen zum Testen durchaus interessant sind, wurde sich für *svm.SVC* entschieden. Zur Evaluation soll die Präzision folgendermaßen definiert sein,

$$\frac{\#\text{true positives}}{\#\text{true positives} + \#\text{false positives}}$$

während die Trefferquote

$$\frac{\#\text{true positives}}{\#\text{true positives} + \#\text{false negatives}}$$

ist.
Die in Tabelle 1 gezeigten Werte wurden unter Verwendung aller 16 Körperteilslängen von allen 11 Personen erreicht, während in den Abbildungen 3 und 4 zur besseren Darstellung nur zwei Körperteile von zwei Personen verwendet wurden: der Nacken und die Schultern.

Person	Präzision	Trefferquote
1	1.00	1.00
2	0.99	1.00
3	0.98	0.98
4	1.00	0.99
5	0.99	1.00
6	1.00	1.00
7	0.90	0.97
8	0.98	0.99
9	0.99	0.99
10	0.99	0.99
11	0.99	0.86
Durchschnitt	0.98	0.98

Tab. 1: Präzision und Trefferquote der SVM pro Person und im Durchschnitt

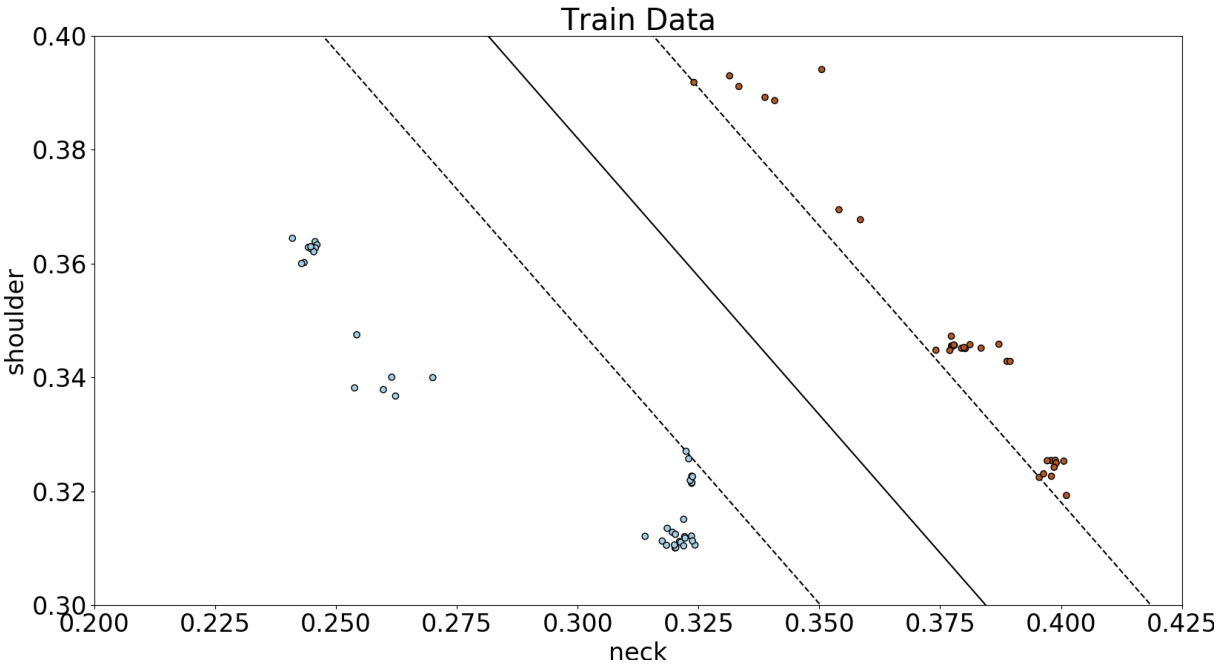

Abbildung 3: Schulter- und Nackenlänge zweier Personen, welche als Trainingsgrundlage dienten. Die zwei gestrichelten Geraden verlaufen in Richtung der Support Vektoren, die durchgezogene Gerade ist die Trennlinie nach dem Trainingsvorgang, gezeichnet mit matplotlib

Es konnte ein relativ großer Abstand zwischen der Trennlinie und den Stützvektoren gefunden werden, was sich in der guten Testperformance in Abbildung 4 zeigt.

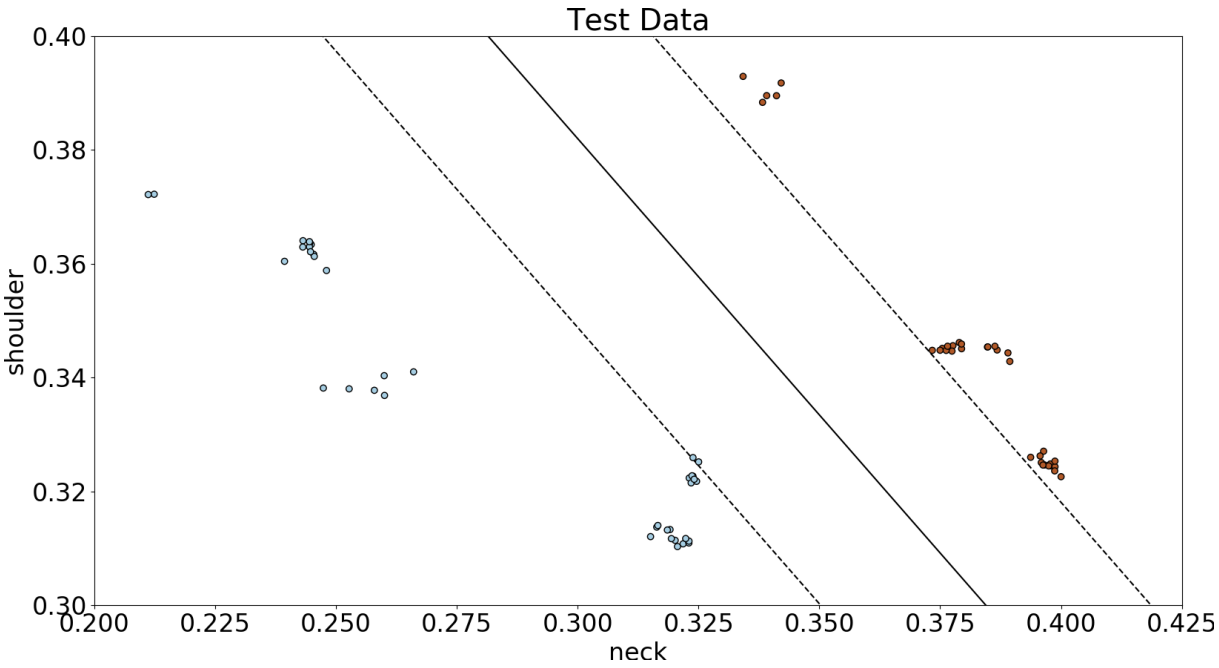

Abbildung 4: Schulter- und Nackenlänge der selben zwei Personen, aber zu anderen Zeitpunkten der Messung (unbekannt für die SVM), gezeichnet mit matplotlib

Das Training der SVM benötigte $2,6$ Sekunden, das Zuordnen von 13135 Elementen dauerte $4,2$ Sekunden.
Sowohl die Verwendung der SVM als auch die Methoden zur Darstellung der Abbildungen 3 und 4 sind in der Datei **svm.py** implementiert.

2.5 Decision Tree

Decision Trees sind ein schon über 50 Jahre altes Konzept [5], welches gegenüber der Support Vector Machine sehr intuitiv funktioniert. Aufgrund des bereits guten Erfolges der SVM in Kapitel 2.4 bietet sich der Entscheidungsbaum als weiterer Kategorisierungskandidat an, da auch dieser mit linear separierbaren Datensätzen sehr präzise und vor allem effizient umgehen kann.

2.5.1 Funktionsweise

Die Klassifizierung eines neuen Datenpunkts geschieht durch ein einfaches Abarbeiten mehrerer Bedingungsabfragen, welche im Trainingsvorgang gelernt wurden. Diese Abfragen werden in einen Binärbaum zusammengefasst, d.h. jeder Knoten hat nur maximal zwei Folgeknoten. Eine Klasse wurde erkannt, wenn ein Pfad von Abfragen bis zu einem Blatt führt.
Da es viele Algorithmen zum Erstellen eines Entscheidungsbaums gibt (s. z.B. *ID3* [6]) und

diese mitunter relativ kompliziert sind, soll im Folgenden ein intuitiver Ansatz zum Aufbau eines Decision Trees gezeigt werden.

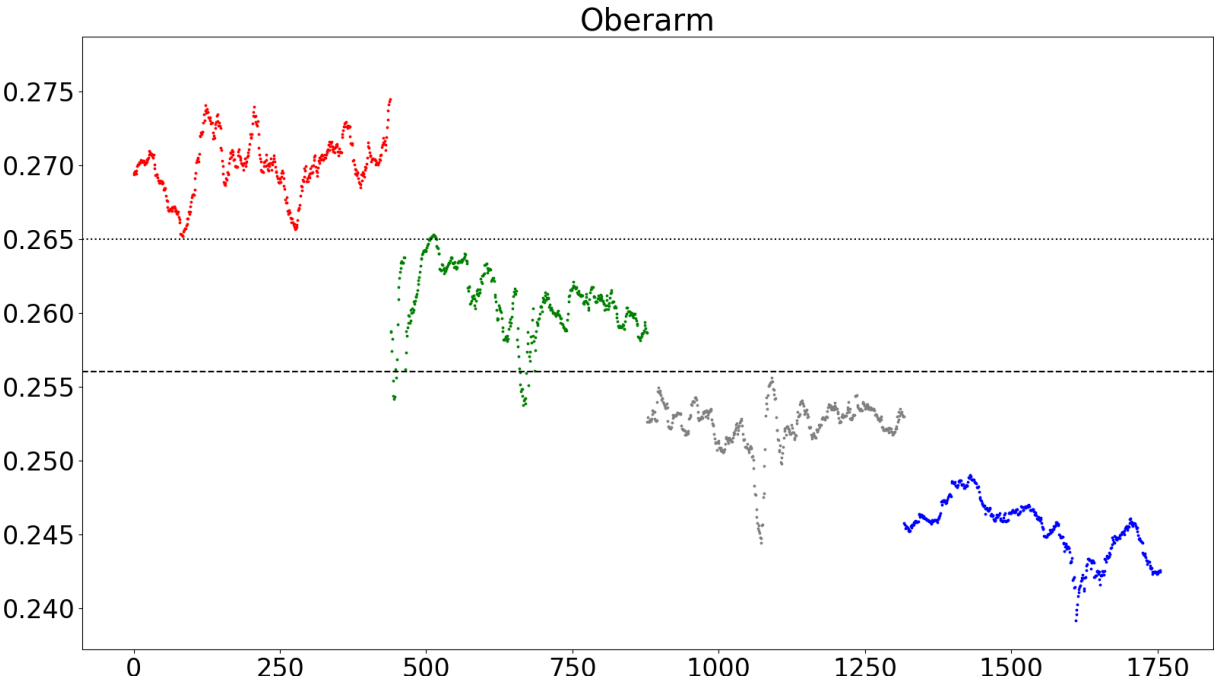

Abbildung 5: Unterarmlängen von vier Personen und mögliche Trennlinien, gezeichnet mit matplotlib

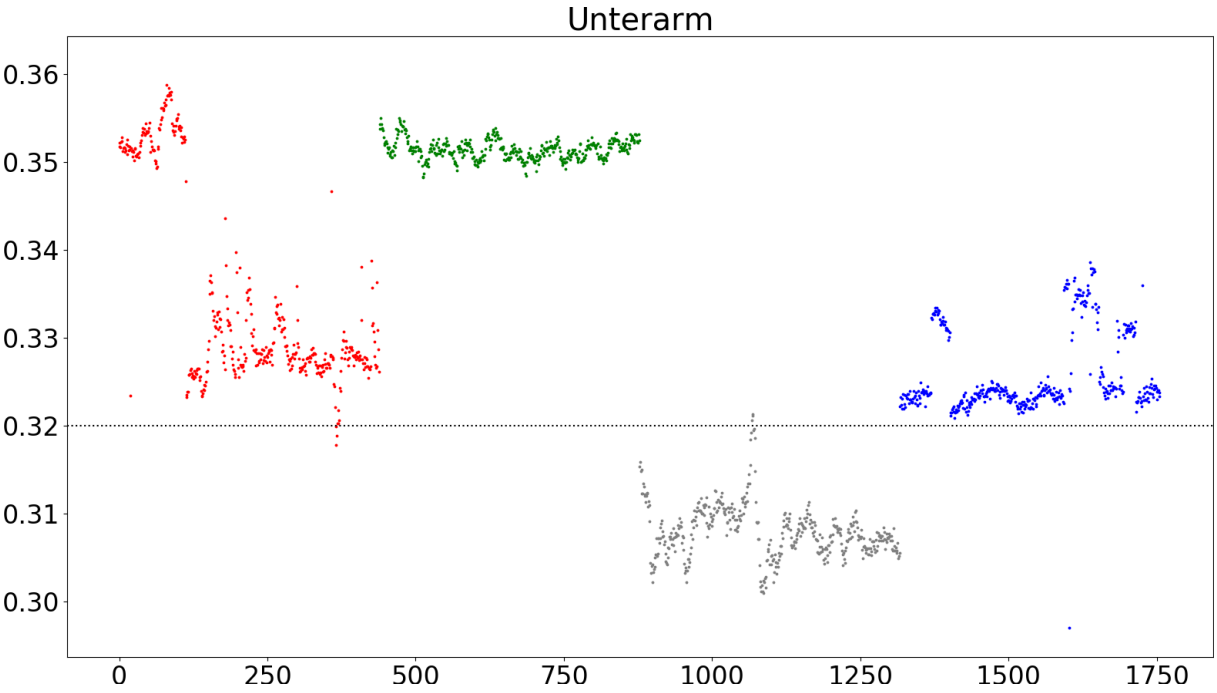

Abbildung 6: Oberarmlängen von vier Personen und mögliche Trennlinie, gezeichnet mit matplotlib

Als Inputs für unseren Entscheidungsbaum seien die Länge des Unterarms (Abb. 6) und die Länge des Oberarms (Abb. 5) von vier Personen (rot,grün,grau,blau) gegeben. Nun versucht man, den Wert zu finden, anhand dessen sich der Datensatz am besten aufteilen lässt. Die genaue Strategie hierbei variiert je nach verwendetem Algorithmus, in diesem Fall könnte an zum Beispiel versuchen jeden Datensatz mit $2n$ Gruppen in kleinere Datensätze mit n Gruppen aufzuteilen. Dies geschieht mit der horizontalen Trennlinie (gestrichelt) bei 0.32 in Abbildung 5. Die beiden Gruppen grau und blau werden durch diese Trennung komplett von den restlichen Gruppen isoliert. Es gilt also: ist der Oberarm kürzer als ca. $0,256$, so handelt es sich wahrscheinlich um Gruppe grau oder blau, sonst um Gruppe rot oder grün.
Um nun noch zwischen diesen Gruppen unterscheiden zu können, betrachtet man noch zusätzlich die Unterarmlänge. Die Trennlinie in Abbildung 6 ist hierbei ein guter Kandidat für die Unterteilung zwischen grau und blau.
Analog wird das selbe Verfahren für die verbleibenden Gruppen rot und grün durchgeführt: Diese lassen sich durch die gepunktete Trennlinie in Abbildung 5 bei 0.265 gut separieren.
Die Abfragen unseres Baumes sind dann:

```
if Oberarm < 0.256
   if Unterarm < 0.32
      grau erkannt
   else
      blau erkannt
else
   if Oberarm > 0.265
      rot erkannt
   else
      grün erkannt
```

Code 3: einfacher Entscheidungsbaum

Selbst dieser sehr simple Entscheidungsbaum erreicht bei ihm unbekannten Testdaten eine Genauigkeit von 0.98 (2 Inputfeatures, 4 Personen). Es sei jedoch angemerkt, dass die tatsächliche Vorgehensweise beim Aufbauen eines Entscheidungsbaumes nicht so ”primitiv” wie in diesem Beispiel verläuft, das selbe Problem jedoch ähnlich wie bei einem intuitiven Verfahren löst.

2.5.2 Umsetzung

sklearn bietet auch für Entscheidungsbäume eine Klasse: *tree.DecisionTreeClassifier*. Dieser unterstützt mehrere Strategien zum Aufbauen des Baumes (z.B. maximaler Informationsgewinn,

s. [6]), wobei in diesem Fall alle unterstützten Strategien gleich präzise sind.
Eines der Probleme eines Decision Trees ist, wenn dieser schlecht generalisiert, d.h. wenn er mit noch nicht gesehenen Daten schlecht umgeht. Dies geschieht vor allem dann, wenn der Baum sehr tief wird. Um dies zu verhindern sollte die niedrigste Tiefe gewählt werden, die noch akzeptable Ergebnisse liefert.

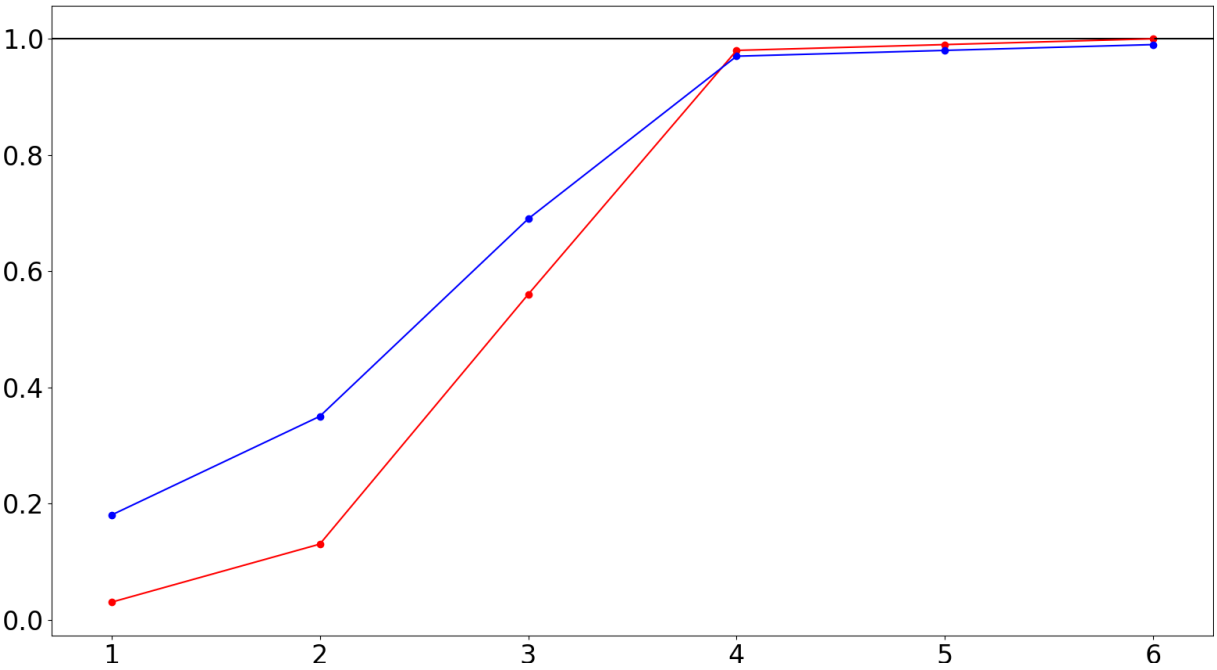

Abbildung 7: Tiefe des Entscheidungsbaums gegen Präzision und Trefferquote

Wie in Abbildung 7 zu sehen ist, erreicht der Baum eine Präzision von $1,00$ bei einer Trefferquote von $0,99$ bereits bei einer Tiefe von 6 Ebenen. Wenn diese Limitierung nicht durchgeführt wird (durch das Setzen des Parameters *max_depth*), entsteht ein Baum der Tiefe 14. Die Trainingszeit beträgt hierbei nur ca. 450 Millisekunden, während die Klassifizierung von 13135 Testdatenpunkten nur 15 Millisekunden benötigte.

2.6 Fazit

Beide Methoden der Kategorisierung (SVM,Entscheidungsbaum) stellten sich als enorm performant heraus. Sie erreichten Präzisionen von über 0.99 bei 11 Personen mit jeweils 16 Körperfeatures.
Der größte Unterschied zwischen beiden Algorithmen ist die Laufzeit zum Klassifizieren und Trainieren. Der Entscheidungsbaum war ca. 6 mal so schnell fertig trainiert und klassifizierte ca.

250 mal schneller als die SVM.

3 Zusammenfassung

Das Projekt stellte sich als zeitlich anspruchsvoll heraus, wäre also wahrscheinlich auch zur Bearbeitung in einer Zweiergruppe geeignet gewesen. Dennoch konnte der Zeitplan eingehalten werden.

In einige Teilbereiche des Machine Learnings wurde sich tief eingearbeitet (SVM,Decision Trees) und es konnten zwei hochperformante Kategorisierer gebaut werden. Der in dieser Arbeit nicht erwähnte Ansatz des Deep Learnings stellte sich als ungeeignet für diese Aufgabe heraus. Zur weiteren Verwendung im behandelten Anwendungsfall wird aufgrund seiner Laufzeitperformanz der Entscheidungsbaum empfohlen.

Die Kinect Daten wurden nach ihrer Bearbeitung als gut geeignet für dieses Projekt befunden.

Literaturverzeichnis

[1] Wie funktioniert kinect? http://www.xida.de/2012/01/31/wie-funktioniert-kinect/. Zugriff: 21.06.2018.

[2] Jamie Shotton, Andrew Fitzgibbon, Mat Cook, Toby Sharp, Mark Finocchio, Richard Moore, Alex Kipman, and Andrew Blake. Real-time human pose recognition in parts from single depth images. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*. IEEE, 2011.

[3] History of svms. http://www.svms.org/history.html. Zugriff: 21.06.2018.

[4] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*. ACM, 1992.

[5] Wei-Yin Loh. Fifty years of classification and regression trees. *International Statistical Review*, 82(3), 2014.

[6] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.