Andreas Ernst Festl, BSc

Erstellen von Referenzzyklen auf Basis statistischer Untersuchungen

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

Masterstudium Operations Research und Statistik

eingereicht an der

Technischen Universität Graz

Betreuer:
Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Herwig Friedl

Institut für Statistik

Graz, im Februar 2015

EIDESSTATTLICHE ERKLÄRUNG

AFFIDAVIT

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Datum/Date	Unterschrift/Signature

Kurzbeschreibung

Will man einen Automotor entwickeln oder verbessern, so ist es unumgänglich zu wissen wie dieser Motor verwendet werden wird. Aus vielen Stunden an Messungen an sich bereits im Betrieb befindlichen Motoren will man deshalb kurze *Referenzzyklen* bestimmen, die alle wichtigen Charakteristika der Messungen enthalten. Ein großer Teil der Arbeit beschäftigt sich deshalb mit der Charakterisierung von Messungen. Zu diesem Zweck wird zunächst ein Algorithmus zur Mustersuche vorgestellt, der typische, wiederkehrende Abschnitte in den Messungen finden soll. Dieser stellt sich jedoch wegen zu hoher Laufzeit als ungeeignet heraus.

Als geeignete Repräsentation stellt sich schließlich die auf der *Symbolic Aggregate approXimation* (SAX) basierende *bag-of-patterns* Repräsentation heraus, die Häufigkeiten von bestimmten, in der Zeitreihe vorkommenden Strukturen beschreibt. Diese wird in verschiedenen Varianten getestet und die jeweilige Performance mit Support-Vector-Machines bewertet. Auch eine Methode zur simultanen Verwendung von mehreren Messkanälen wird vorgestellt. Die mit dieser Methode repräsentierten Messungen werden schlussendlich mit verschiedenen Verfahren geclustert und es werden Möglichkeiten präsentiert mit Hilfe der gefundenen Cluster Referenzzyklen zu erzeugen.

Abstract

To develop or improve car engines, it is necessary to know how the engine is going to be used. Hence, one wants to create short *reference cycles* from many hours of measurements that contain all their important aspects. A large part of the thesis therefore deals with the characterization of measurements. At first, an algorithm for pattern discovery which should find typical, recurring patterns in the measurements is presented. However, this algorithm is found to be inappropriate for our present situation because of its very long runtime. Eventually, the *bag-of-patterns* (BOP) representation based on the *Symbolic Aggregate approXimation* (SAX) is found to be eligible. This representation describes a time series by counting the occurrences of certain structures. Multiple varieties of the BOP representation are tested and their respective performance is evaluated using support vector machines. Additionally an adaption of BOP for taking multiple measurement channels into account is presented. The measurements, represented with said adaption, are finally clustered by several different techniques, and possibilities to create reference cycles from the found clusters are shown.

Inhaltsverzeichnis

1. Einleitung

1.1. Aufgabenstellung und Aufbau der Arbeit

Die Masterarbeit wurde in Zusammenarbeit mit der AVL List GmbH erstellt. AVL ist ein Unternehmen, dass sich vor allem mit der Entwicklung und dem Test von Antriebssystemen beschäftigt und auch dazugehörige Prüf- und Messsysteme entwickelt. Will man aber einen Automotor entwickeln oder verbessern, so ist es unumgänglich zu wissen wie dieser Motor verwendet werden wird. Dies wird durch das so genannte *Lastkollektiv* beschrieben. Dieses soll die Gesamtheit der Nutzungs- und Belastungsaspekte denen der Motor ausgesetzt war oder sein wird widerspiegeln. Das Lastkollektiv wird anhand von Messungen an sich bereits im Betrieb befindlichen Motoren bestimmt. Diese Messungen sind typischerweise jeweils mehrere Stunden lang und enthalten viele verschiedene Messkanäle wie etwa die Fahrzeuggeschwindigkeit, das Motordrehmoment, die Motordrehzahl und viele mehr. Es sind auch ausreichend viele Messungen zur Beschreibung des Lastkollektivs vorhanden, es stellt sich vielmehr das Problem aus dieser großen Menge wenige Parameter zu extrahieren, mit denen die Menge der Messungen adäquat beschrieben werden kann. Idealerweise wird die Menge der Messungen aber nicht nur durch diese einzelnen Werte beschrieben, sondern es werden zusätzlich *Referenzzyklen* bestimmt. Ein Referenzzyklus ist dabei eine ausgewählte oder künstlich erzeugt Messung, die die Eigenschaften aller vorhandenen Messungen in sich trägt, diese Menge also komprimiert repräsentiert. Im Idealfall reicht es dann, in weiterer Folge nur mehr die Referenzzyklen als Stellvertreter für die Menge der Messungen zu betrachten. Das hat enorme Vorteile, sei es auf dem Prüfstand, wo man den Motor einem realistischen Betrieb unterziehen will und dafür nun den Referenzzyklus als Input verwenden kann, oder bei der Interpretation der Messungen durch Experten, die nur mehr die als Referenzzyklus ausgewählten Messungen betrachten müssen.

In dieser Masterarbeit soll nun eine Methodik entwickelt werden um solche Referenzzyklen zu finden, beziehungsweise diese aus Teilen der Originalmessungen zu erzeugen. Der Aufbau der Arbeit folgt der tatsächlichen Vorgehensweise, es wird also immer eine getestete Methode vorgestellt und dann die mit dieser Methode erzielten Ergebnisse präsentiert.

Zunächst wird näher auf die Beschaffenheit und Zusammensetzung der Messungen die für das Testen der verschiedenen Methoden verwendet wurden eingegangen. In Kapitel 2 wird versucht Muster in (einem Kanal) der Messungen zu finden um die Messungen über die gefundenen Muster zu charakterisieren. Im darauffolgenden Kapitel wird die „bag-of-patterns“ Repräsentation für Zeitreihen vorgestellt, zunächst auch hier für nur einen Messkanal. Die Performance dieser Repräsentation wird danach mit Hilfe von Support-Vector-Machines bewertet. Nachdem eine durchwegs gute Performance erzielt wird, wird die bag-of-patterns Repräsentation für die Verwendung von mehreren Messkanälen angepasst. In Kapitel 6 werden drei verschiedene Verfahren zum Clustern von Daten vorgestellt die in den darauffolgenden Kapiteln auf die Messungen angewandt werden. Zuletzt werden Messungen aus den Clusterzentren als Referenzzyklen ausgewählt, sowie eine Metho-

de vorgestellt mit der Referenzzyklen aus Teilen der Originalmessungen zusammengesetzt werden können.

1.2. Beschaffenheit der Daten

Gegeben sind Messungen die an verschiedenen LKW-Motoren durchgeführt wurden. Gemessen wurde in verschiedenen Fahrsituationen im realen Betrieb. Diese Betriebsmodi waren:

- Fahrten auf Autobahn und Bundesstraße
- Fahrten über eine Passstrecke
- Fahrten eines Müllautos
- Fahrten im Stadtverkehr
- Fahrten im Verteilervekehr (also großteils Stadtverkehr und Bundesstraße gemischt)

Jede Messung besteht dabei aus verschiedenen Messkanälen, deren Anzahl und Auswahl sich jedoch von Messung zu Messung unterscheiden kann. Die Messkanäle sind teilweise zeitindiziert, teilweise kommen aber auch andere Indizierungen, wie etwa die Position der Kurbelwelle, zum Einsatz. Typische Messkanäle wären etwa: die Fahrzeuggeschwindigkeit, das Motordrehmoment, die Motordrehzahl, Umgebungsdruck, aktuelle Höhe über dem Meeresspiegel (allgemeiner: GPS-Daten), Positionen des Gas- und Bremspedals und viele weitere. In Abbildung 1.1 ist exemplarisch ein Ausschnitt des Messkanals „Geschwindigkeit“ einer Messung dargestellt.

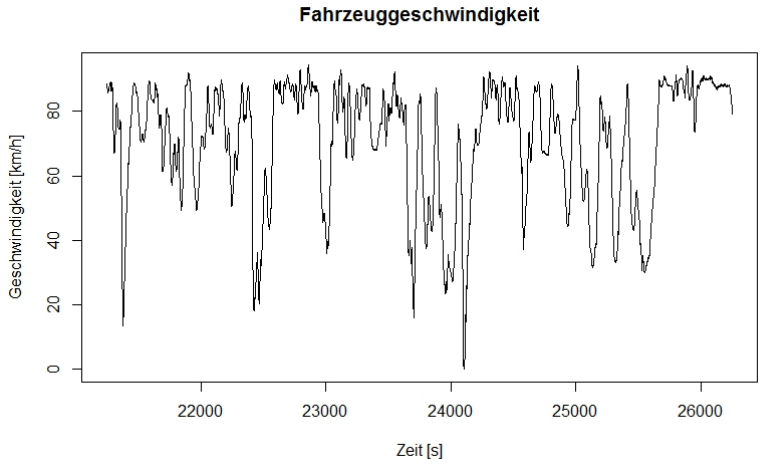


Abbildung 1.1.: Ausschnitt aus einer Messung der Fahrzeuggeschwindigkeit

Die zeitliche Länge der Messungen variiert sehr stark, beträgt aber meist mindestens vier Stunden (14400s), da kürzere Messungen als nicht aussagekräftig betrachtet werden. Gemessen wurde mit einer Auflösung von 1Hz, ein Punkt der Zeitreihe der Messung entspricht also einer Sekunde.

2. Charakterisierung der Messungen durch vorkommende Muster

Die erste Idee war es, Algorithmen zur Mustererkennung (im Sinne des englischen „pattern discovery“ und nicht des „pattern detection“, die beide als „Mustererkennung“ übersetzt werden) zu verwenden. Hat man damit Muster in der Messung gefunden, so könnten Messungen gesucht werden welche die gleichen Muster enthalten, man könnte bestimmen wie viel Prozent einer Messung aus einem bestimmten Muster bestehen und vieles mehr. Um Muster zu erkennen, muss man jedoch Stücke von Zeitreihen miteinander vergleichen können, man braucht also ein Distanz- oder Ähnlichkeitsmaß. Für zwei zu vergleichende Stücke welche die gleiche Länge besitzen, kann man als Distanzmaß offensichtlich jede von einer Vektornorm induzierte Metrik benutzen, da man die Stücke als Vektoren auffassen kann, deren Dimension der Länge der Stücke entspricht. Die „Standard“-Metrik ist dabei die euklidische Metrik:

Beispiel. [Euklidische Distanz] Gegeben seien zwei Stücke $A = (a_1, \ldots, a_n)$ und $B = (b_1, \ldots, b_n)$ einer Zeitreihe. Der euklidische Abstand zwischen diesen beiden Stücken ist:

$$D_{eukl}(A, B) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}$$

$\Diamond$

Probleme treten jedoch dann auf, wenn man zulassen möchte, dass Muster auch unterschiedliche Längen haben dürfen, bzw. wenn Teile des Muster gestaucht oder gestreckt sind. Eine Möglichkeit mit diesen Situationen umzugehen, liefert das Dynamic Time Warping, welches nun vorgestellt werden soll.

2.1. Dynamic Time Warping

Dynamic Time Warping (DTW) ist ein bekannter Algorithmus um Ähnlichkeiten in zeitlichen Messungen zu finden, wobei sich die ähnlichen Teile in ihrer Länge unterscheiden. Die prominenteste Anwendung ist vielleicht die Spracherkennung; Worte werden von verschiedenen Menschen unterschiedlich schnell und mit unterschiedlicher Betonung ausgesprochen. Der DTW-Algorithmus verwendet nun dynamische Programmierung um Teile die sich entsprechen einander zuzuordnen [Wendemuth, 2004]. Das funktioniert nach dem folgenden Prinzip:

Es seien zwei Stücke einer Zeitreihe gegeben, die nun aber unterschiedliche Längen haben dürfen. Seien das $A = (a_1, \ldots, a_n)$ und $B = (b_1, \ldots, b_m)$. Dann muss eine 1-dimensionale Metrik fixiert werden, hier die euklidische Metrik. Das liefert eine Distanzmatrix $D = (D_{ij}), \quad i = 1, \ldots, n, \quad j = 1, \ldots, m$, der zu vergleichenden Stücke. Der Eintrag D_{ij} entspricht dabei dem Abstand zwischen a_i und b_j, hier also $D_{ij} = \sqrt{(a_i - b_j)^2}$.

Nun kann man verschiedene Randbedingungen fixieren, die spezifizieren, welche Punkte der beiden Stücke miteinander verglichen werden dürfen. Dann sucht der DTW-Algorithmus aus jeder Zeile und Spalte der Distanzmatrix zumindest ein Element aus, und zwar so, dass die Gesamtsumme der ausgewählten Matrixeinträge möglichst klein ist und alle Randbedingungen eingehalten werden. Wird das Element D_{ij} gewählt, so bedeutet das, dass der Punkt a_i aus dem ersten Stück dem Punkt b_j aus dem zweiten Stück entspricht und mit diesem verglichen wird. Der DTW-Algorithmus löst also ein kürzeste-Wege-Problem und such einen Weg von einer Seite der Distanzmatrix auf die andere. Die Kosten des Weges (also die Summe der mit den Schrittkosten gewichteten Elemente) liefert dann ein Maß für den Abstand der zwei Stücke A und B. Will man diesen Abstand mit jenem von Stücken unterschiedlicher Länge vergleichen, muss man die Wegkosten normieren. Die genaue Normierung hängt vom Schrittmuster ab, und es gibt auch Schrittmuster für die keine Normierung möglich ist. Meist ist der Normierungsfaktor der Kehrwehrt der Summe der Schrittkosten oder der Kehrwert der Summe der Längen der beiden Stücke.

Beispiele für sinnvolle Randbedingungen die verwendet werden können sind:

- Anfang und Ende der zwei Stücke entsprechen müssen sich entsprechen
- Zeitlich benachbarte Punkte von A dürfen nur zeitlich benachbarten Punkten von B entsprechen. In diesem Fall sucht der DTW-Algorithmus den kürzesten Weg von D_{11} nach D_{nm} durch die Distanzmatrix.
- Üblicherweise werden so genannte *Schrittmuster* definiert. Diese definieren die Nachbarschaft von Punkten in der Matrix, also die welche Schritte (horizontal/vertikal/-diagonal) erlaubt sind und deren Kosten.
- Weitere Randbedingungen können beispielsweise festlegen, dass sich die „Steigung“ des Wegs durch die Matrix nicht zu weit von der Steigung der gedachten Diagonale (von D_{11} nach D_{nm}, also $\frac{m}{n}$) entfernen darf.

Die Berechnung der DTW-Distanz kann in `R`zum Beispiel mit dem `dtw`-Package [Giorgino, 2009] erfolgen, welches auch hier verwendet wurde.

Zur Verdeutlichung betrachten wir ein kleines Beispiel:

Beispiel. [DTW-Minibeispiel] Gegeben seien die zwei Zeitreihen $A = (1, 2, 3, 2, 1)$ und $B = (1, 3, 1.5, 1)$, dargestellt in Abbildung 2.1.

Wir fixieren die euklidische Metrik als zugrunde liegendes Distanzmaß und erhalten damit folgende Distanzmatrix:

$$\begin{pmatrix} 0 & 1 & 2 & 1 & 0 \\ 2 & 1 & 0 & 1 & 2 \\ 0.5 & 0.5 & 1.5 & 0.5 & 0.5 \\ 0 & 1 & 2 & 1 & 0 \end{pmatrix} \tag{2.1}$$

Als Randbedingungen fixieren wir hier folgendes: Anfangs- und Endpunkte der beiden Stücke sollen sich entsprechen. Das verwendete Schrittmuster ist „symmetric2“ (Name aus dem `dtw`-Package übernommen). In diesem Schrittmuster sind alle Schritte erlaubt, die den Zeitindex in einer der Zeitreihen vergrößern, also Schritte in die Richtungen S (vertikal),

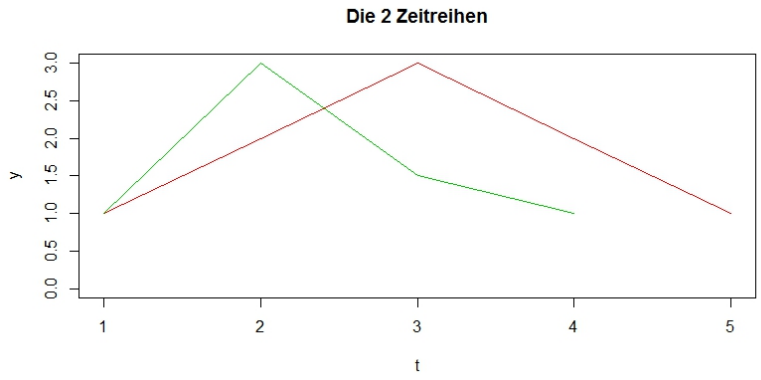


Abbildung 2.1.: Die 2 Zeitreihen enthalten das selbe Muster, jedoch zeitlich verzerrt. Sie sollen mit DTW verglichen werden

O (horizontal) und SO (diagonal). Damit der diagonale Schritt nicht als „Abkürzung“ missbraucht werden kann, sind die Kosten für diesen doppelt so groß wie für die anderen beiden Schritte.

In der Distanzmatrix (2.1) ist der so gefundene kürzeste Weg grün markiert. Die unnormierten Kosten erhalten wir gemäß dem Schrittmuster als:

$$0 + 1 \cdot 1 + 2 \cdot 0 + 2 \cdot 0.5 + 2 \cdot 0 = 2.$$

Als Normierung kann man bei diesem Schrittmuster die Summe der Längen der beiden Stücke verwenden, womit sich die normierte Distanz der Stücke unter DTW als 2/9 ergibt. ◊

Als zweites Beispiel betrachten wir zwei unterschiedlich lange Stücke von mit weißem Rauschen gestörten Sinusschwingungen mit gleicher Frequenz, aber unterschiedlicher Phase. Diese Stücke sind in Abbildung 2.2 zu sehen.

Wir wollen diese Stücke nun unter folgenden Randbedingungen einander zuordnen (Namen der Schrittmuster aus dem `dtw`-Package:

1. Das Schrittmuster „rigid“. Dieses Schrittmuster erlaubt nur diagonale Schritte in der Distanzmatrix, das heißt, wir suchen eine 1:1 Abbildung der Punkte der beiden Stücke, wobei benachbarte Punkte benachbart bleiben. Diese starke Einschränkung macht natürlich nur Sinn, wenn man Anfangs- und Endpunkte nicht fixiert. In diesem Fall wird dann das optimale Alignment der beiden Stücke gefunden.

2. Das Schrittmuster „symmetric2“. Gleiches Schrittmuster wie im DTW-Minibeispiel.

3. Schrittmuster „symmetric2“ mit Itakura Window. Das selbe Schrittmuster wie in 2), allerdings darf sich der Pfad durch die Distanzmatrix nicht zu weit von der Diagonale entfernen. Damit wird verhindert, dass Punkte deren Indices sehr weit außereinander liegen, einander zugeordnet werden.

Die gefundenen Zuordnungen, sowie die resultierende Distanz sind in Abbildung 2.3 ersichtlich.

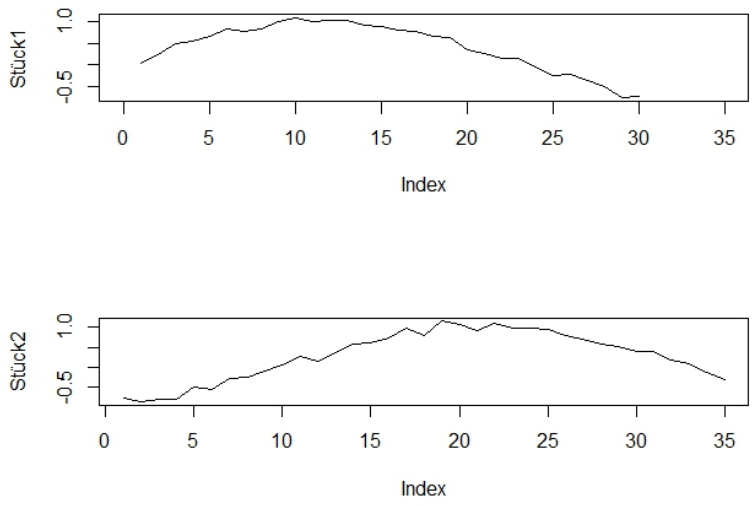


Abbildung 2.2.: Ausschnitte von zwei verrauschten Sinusschwingungen mit den Längen 30 (oben) und 35 (unten)

2.2. Der Algorithmus zur Mustererkennung

Zum Finden von Mustern wurde ein probabilistischer Algorithmus verwendet, der von Catalano et al. [2006] beschrieben wird. Dieser funktioniert so:

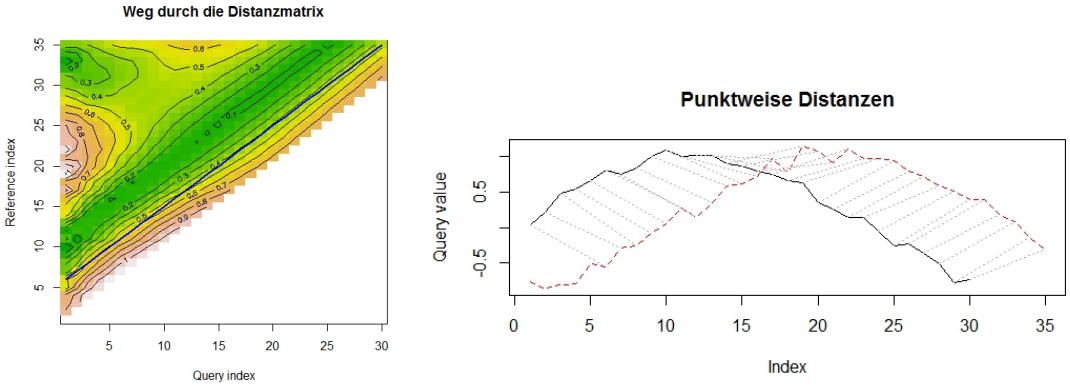
Zunächst definiert man die *window size* w und die *subwindow size* $\bar{w} < w$. Dann zieht man aus der Zeitreihe eine bestimmte Anzahl von Fenstern, wobei ein Fenster aus w aufeinander folgenden Punkten $F_1, \ldots, F_w$ besteht. Jedes dieser Fenster wird nun in $w-\bar{w}+1$ Subwindows S_i^F eingeteilt, wobei $S_1^F = F_1, \ldots, F_{\bar{w}}$, $S_2^F = F_2, \ldots, F_{\bar{w}+1}$, $\ldots$, $S_{w-\bar{w}+1}^F = F_{w-\bar{w}+1}, \ldots, F_w$.

Nun werden jeweils alle zu einem Fenster gehörenden Subwindows einer von zwei Mengen zugeordnet; entweder der *Kandidatenmenge* oder der *Vergleichsmenge*. Wie viele Fenster in jede Menge kommen, ist ein Parameter der vom Benutzer bestimmt werden muss.

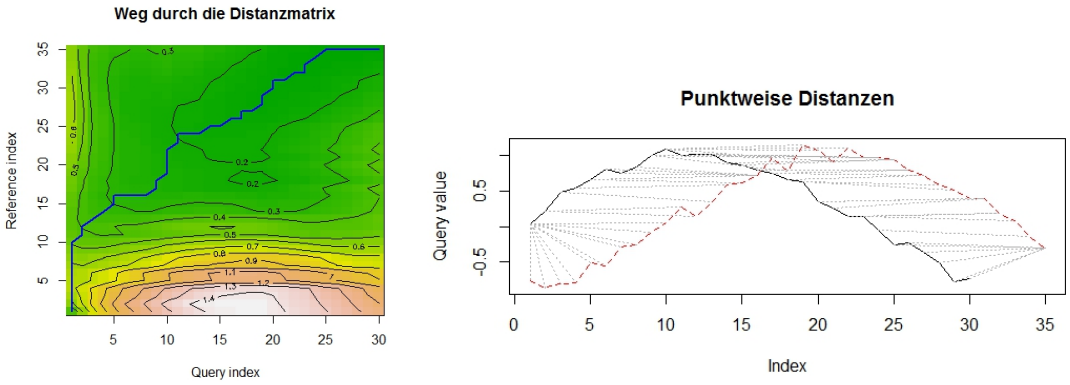
Im nächsten Schritt wird für jedes Subwindow der Kandidatenmenge die Distanz zu allen Subwindows der Vergleichsmenge berechnet und das arithmetische Mittel der k-besten Distanzwerte (unter DTW sind das die k kleinsten) als *mittlere beste Distanz des Subwindows* gespeichert. Auch der Parameter k ist hierbei noch zu definieren.

Hat man für alle Subwindows der Kandidatenmenge ihre mittlere beste Distanz berechnet, gilt es, jene Subwindows zu identifizieren, für die dieser Wert besonders gut (DTW: besonders klein) ist. Dazu simuliert man die Verteilung der mittleren besten Distanzen von Subwindows die sicher kein Muster enthalten. Um solche musterfreien Subwindows zu erhalten, generiert man ein *Noise-Window* der Länge w, indem man w Punkte der Zeitreihe zufällig zieht. Insbesondere sind die Punkte nicht notwendig aufeinanderfolgend, weshalb davon ausgegangen werden kann, dass das resultierende Fenster kein Muster der originalen Zeitreihe enthält. Das erzeugte Noise-Window unterteilt man nun analog zu oben in seine Subwindows und berechnet die mittlere beste Distanz aller Subwindows der Kandidatenmenge zu den Noise-Subwindows, was eine Stichprobe der Verteilung der mittleren besten

Schrittmuster „rigid“ ergab eine normalisierte Distanz von 0.43:



Schrittmuster „symmetric2“ ergab eine normalisierte Distanz von 0.14:



Schrittmuster „symmetric2“ mit Itakura Window ergab eine normalisierte Distanz von 0.35:

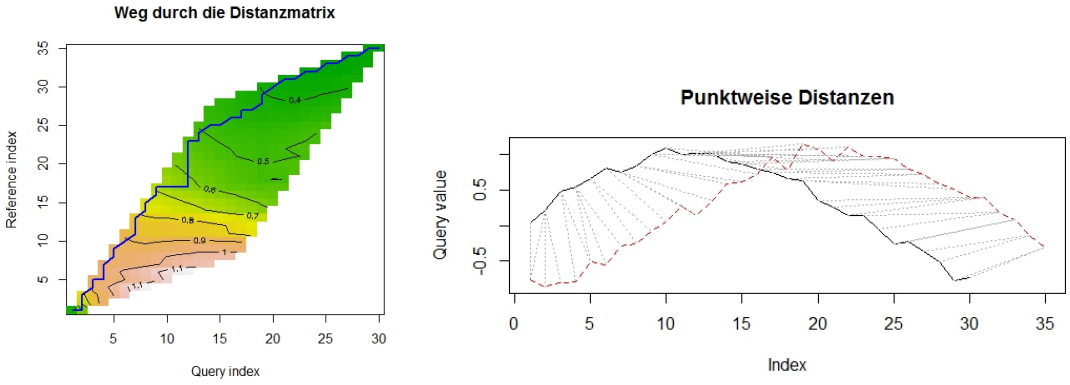


Abbildung 2.3.: DTW der beiden Sinusschwingungen unter verschiedenen Randbedingungen (Schrittmuster und Entfernung des Pfades von der Diagonale).

Distanzen von musterfreien Subwindows liefert.

Nun definiert man ein Signifikanzniveau α, schätzt aus der Noise-Stichprobe das entsprechende Quantil, und entfernt alle Subwindows mit zu schlechter mittlerer bester Distanz aus der Kandidatenmenge. Unter DTW würde man also das α-Quantil schätzen und alle Subwindows mit mittlerer bester Distanz größer dem Quantil entfernen. Für Ähnlichkeitsmaße, bei denen ein höherer Wert eine höhere Ähnlichkeit nahelegt (etwa Korrelation), würde man das $(1-\alpha)$-Quantil schätzen und alle Subwindows mit kleinerer Distanz entfernen.

Als nächstes wird dieses Prozedere für eine bestimmte Anzahl von Schritten iteriert, indem eine neue Vergleichsmenge analog zu vorher generiert wird. Die Kandidatenmenge bleibt aber natürlich gleich beziehungsweise kann in jedem Schritt nur verkleinert werden. Die nach Abschluss dieser Prozedur verbleibenden Subwindows enthalten dann Muster.

Zuletzt werden benachbarte oder überlappende Subwindows wieder zu längeren Stücken kombiniert.

2.2.1. Anmerkungen zum Algorithmus

- Die Länge der Subwindows $\bar{w}$ entspricht dem kürzesten Muster, das erkannt werden kann, und die Länge der Windows w dem längsten. (Es sei denn, die zufällig gewählten Kandidatenfenster sind direkt benachbart oder überlappen sich, dann ist auch die Erkennung von einzelnen längeren Mustern möglich.)

- Die Subwindows können, müssen aber nicht, vor dem Vergleich noch geeignet normalisiert werden.

- Der Algorithmus reagiert sehr sensitiv auf Änderungen in den Parametern. Die größten Auswirkungen scheint hier der Parameter k zu haben, der bestimmt, über wie viele Werte das Mittel der Distanzen für ein Subwindow bestimmt wird. Schon kleine Änderungen von k führen dazu, dass sich die mittleren besten Distanzwerte nicht mehr von der Noise-Distribution unterscheiden (k zu groß), oder aber zu viele Subwindows signifikante Distanzwerte haben (k zu klein). Das ist besonders problematisch, da der beste Wert von k von der unbekannten Anzahl der Vorkommen der Muster abhängt (Idealerweise ist k genau die Anzahl der Mustervorkommen in der Vergleichsmenge).

- DTW hat grundsätzlich quadratische Laufzeit. Nimmt man aber die Länge der Subwindows $\bar{w}$ als konstant an, ergibt sich für die DTW-Berechnung ein konstanter Beitrag zum Gesamtalgorithmus. Dieser benötigt $[\#Kandidatenfenster]\cdot[\#Vergleichsfenster]\cdot[\#Iterationen]\cdot(w-\bar{w}+1)^2$ DTW-Aufrufe, was immer noch recht viel ist. Deshalb wurde der Algorithmus mit Hilfe des `R`-Packages `snowfall` parallelisiert und auf 4 Kernen gleichzeitig ausgeführt. Leider zeigte sich trotzdem, dass die zur benötigte Zeit für einen Suchlauf in einer einzelnen Zeitreihe der realen Daten immer mehrere Stunden betrug, sofern Parameter verwendet wurden bei denen nicht gleich alle Muster verworfen wurden.

- Insbesondere auch wegen der hohen Laufzeit von DTW wurde der Algorithmus so implementiert, dass auch andere Ähnlichkeitsmaße einfach zu verwenden sind. Konkret benötigt der Algorithmus nur eine Funktion, die als Parameter zwei Subwindows

übernimmt und einen Ähnlichkeitswert zurückgibt, sowie die Information, ob große oder kleine Werte eine höhere Ähnlichkeit nahelegen.

2.3. Test auf künstlichen und einfachen Daten

Bevor Muster in den realen Daten gesucht werden, wird der Algorithmus auf künstlich erzeugten und besonders einfachen realen Daten getestet.

2.3.1. Test auf künstlich erzeugten Daten

Zur Erzeugung der künstlichen Daten wurde zuerst ein bimodales Muster der Länge 15 erstellt. Dazu wurde der Graph des Polynoms $f(x) = -0.5x^4 + 0.7x^3 + 5.7x^2 - 0.8x + 10$ für $x \in [-3, 4]$ gezeichnet und an 15 gleichmäßig in diesem Intervall verteilten Stützstellen ausgewertet. Der zugehörige x-Wert war dann nicht länger relevant, und die resultierende Zeitreihe wurde mit 1 bis 15 indexiert. Diese 15 Werte wurden dann 19 mal wiederholt, wobei die Instanzen nicht direkt aneinander gereiht wurden, sondern immer eine gewisse Anzahl an Punkten mit dem y-Wert 0 eingefügt wurde. Die Anzahl der Punkte in diesen „0-Stücken“ wurde jedes Mal neu gleichverteilt aus [5,65] gewählt. Die so zusammengesetzte Zeitreihe wurde zuletzt noch mit standard-normalverteiltem weißen Rauschen gestört. Das Endergebnis ist in Abbildung 2.4 zu sehen.

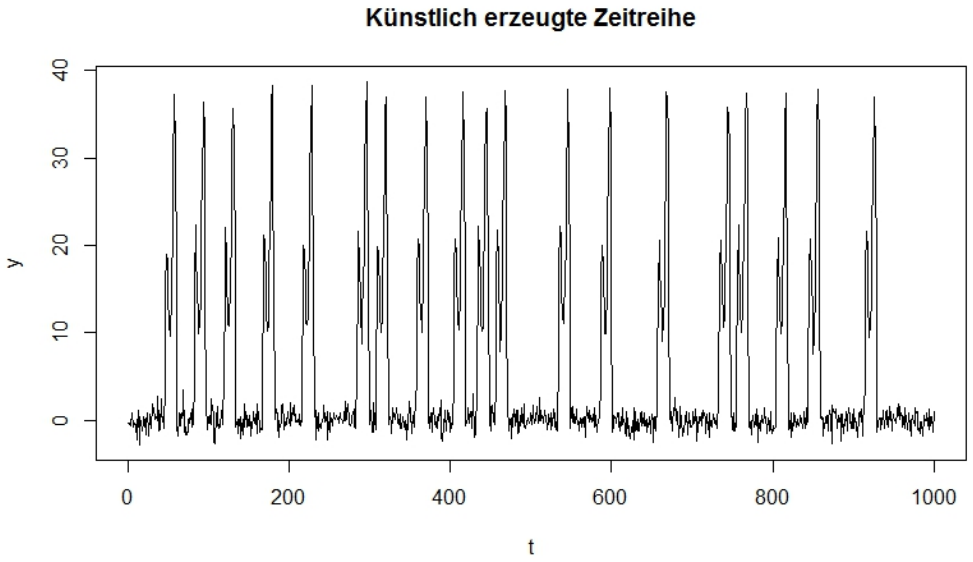


Abbildung 2.4.: Die künstlich erzeugte Zeitreihe mit 19 Instanzen des bimodalen Musters.

Ergebnisse der Experimente:

Beim Experimentieren mit verschiedenen Distanzmaßen und Parametersettings zeigte sich, dass bei diesen Daten der Korrelationskoeffizient als Distanzmaß am besten funktionierte. Es zeigte sich auch, dass die im Algorithmus verwendeten Fenster einen relativ

großen Teil der Daten überdecken müssen, damit der Algorithmus gute Ergebnisse liefert. Konkret konnten erst brauchbare Ergebnisse erzielt werden, wenn die Gesamtlänge der gesampelten Fenster mindestens 50% der Länge der gesamten Zeitreihe ausmachte. Die verwendeten Parameter und Beispiele gefundener Muster finden sich in Anhang A.1.

Man sieht, dass der Algorithmus das korrekte Muster (oder zumindest Teile davon) erkennt. Reines weißes Rauschen wurde in diesem Fall nie als Muster erkannt. Die einzelnen Teile entstehen wenn die Kandidatenfenster die Musterinstanz nicht komplett überlappen.

2.3.2. Test auf einfachen reale Daten

Nachdem der Algorithmus auf den künstlich erzeugten Daten recht gute Ergebnisse lieferte, wurde er als nächstes an einfachen realen Daten getestet. Als Ausgangszeitreihe wurde dafür die Messung der Geschwindigkeit eines Müllfahrzeuges in der Länge von 1000s herangezogen. Diese enthält durch das oftmalige Stoppen des Fahrzeugs sehr charakteristische Muster. Um diese Muster noch ähnlicher zu machen wurde die Messung vor der Mustererkennung mit einem gleitendem Mittel der Ordnung 10 geglättet. Die so vorbereiten Daten sind in Abbildung 2.5 zu sehen.

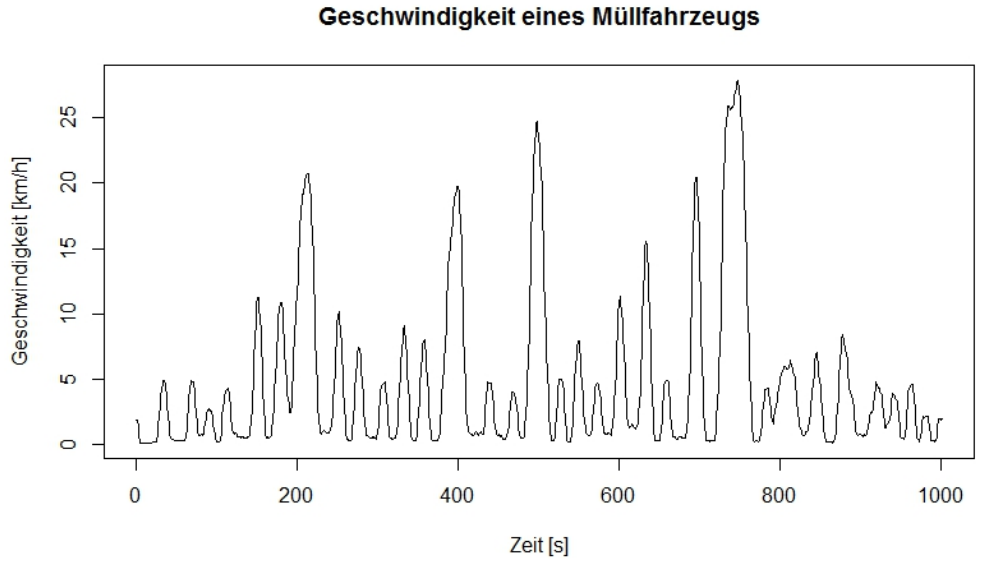


Abbildung 2.5.: Ausschnitt der Geschwindigkeitsmessung eines Müllfahrzeugs; geglättet mit gleitendem Mittel der Ordnung 10.

Zuerst wurde DTW als Distanzmaß verwendet, damit konnte aus den Müllfahrzeug-Daten allerdings nur sehr wenig extrahiert werden. Insbesondere wurden, wenn überhaupt, nur die kleinen Peaks aus Abbildung 2.5 erkannt, die großen, die seltener auftreten, kommen in keinem der erkannten Muster vor. Auch sonst werden eher die Stehzeiten mit relativ konstanten 0 km/h als Muster erkannt (das sie ja in gewissem Sinne auch sind). Einige der gefundenen Muster und die verwendeten Parameter werden in Anhang A.2.1 dargestellt. Insgesamt zeigt sich auch hier, dass ein zu seltenes Auftreten der interessierenden Muster ein Problem darstellt. Dadurch muss/müsste der Algorithmus eine sehr große Anzahl von

Fenstern sampeln, so dass die Muster mit großer Wahrscheinlichkeit überdeckt werden. Dies wird aber von der Laufzeit her sehr schnell inpraktikabel.

Als letzter „Testlauf“ wurde der Algorithmus auf die Müllfahrzeug-Daten mit dem Korrelationskoeffizienten als Distanzmaß angewendet. Hier entsprachen die gefundenen Muster schon eher dem, was auch ein Mensch als Muster in obigen Daten bezeichnen würde. Es wurden sowohl kleine als auch große Peaks, sowie steigende und fallende Flanken mit verschiedenen Steigungen erkannt. (Parameter und Muster in Anhang A.2.2).

Wird im gleichen Ausschnitt der Geschwindigkeitsmessung nach Mustern gesucht ohne die Daten vorher zu glätten, so werden zwar (mit dem Korrelationskoeffizienten) noch immer Teile der typischen Peaks gefunden, allerdings sind die gefundenen Muster durchwegs kürzer und stellen meist nur steigende oder fallende Flanken und nicht den gesamten Peak dar.

Variiert man die Parameter des Algorithmus, so zeigt sich an den einfachen realen Daten, dass besonders die Fenstergröße und die Anzahl der Kandidatenfenster wesentlich ist. Sobald die Anzahl und Länge der gesampelten Fenster groß genug ist, kann man die restlichen Parameter variieren und erhält trotzdem noch ein vergleichbares Ergebnis. Findet sich jedoch nur ein zu kleiner Teil der Messung in den Kandidatenfenstern wieder, so ist der Algorithmus sehr sensitiv. Schon kleine Änderungen in den restlichen Parametern führen dann dazu, dass entweder alle oder gar keine Kandidaten verworfen werden.

Fazit der Tests:

Die Tests an künstlichen und einfachen realen Daten haben gezeigt, dass der Algorithmus mit den richtigen Parametereinstellungen und dem richtigen Distanzmaß durchaus in der Lage ist charakteristische Muster zu finden. Unbedingte Voraussetzung ist aber, dass diese Muster wirklich häufig auftreten und eine ausreichend große Menge an Fenstern gesampelt wird.

2.4. Durchgeführte Experimente auf realen Daten

Nun wurde der Algorithmus auf reale Daten angewendet. Alle Experimente verwendeten dabei den selben Zufallsgenerator mit dem selben Seed. Die Muster wurden in einem Ausschnitt der Zeitreihe der Fahrzeuggeschwindigkeit aus Abschnitt 1.2 gesucht; selbiger ist in Abbildung 2.6 dargestellt. Subwindows wurden nicht normalisiert. Es wurde gestoppt, wenn sich die Anzahl der Kandidaten-Subwindows in vier aufeinanderfolgenden Schritten nicht geändert hatte. Folgende Distanzmaße wurden getestet:

1. DTW
2. 1-Norm
3. 2-Norm
4. Summe aus Mittelwert und Varianz der punktweisen (euklidischen) Distanzen, also $mean(d_1, \ldots, d_n) + var(d_1, \ldots, d_n)$ mit $d_i = \sqrt{(a_i - b_i)^2}$
5. Korrelationskoeffizient

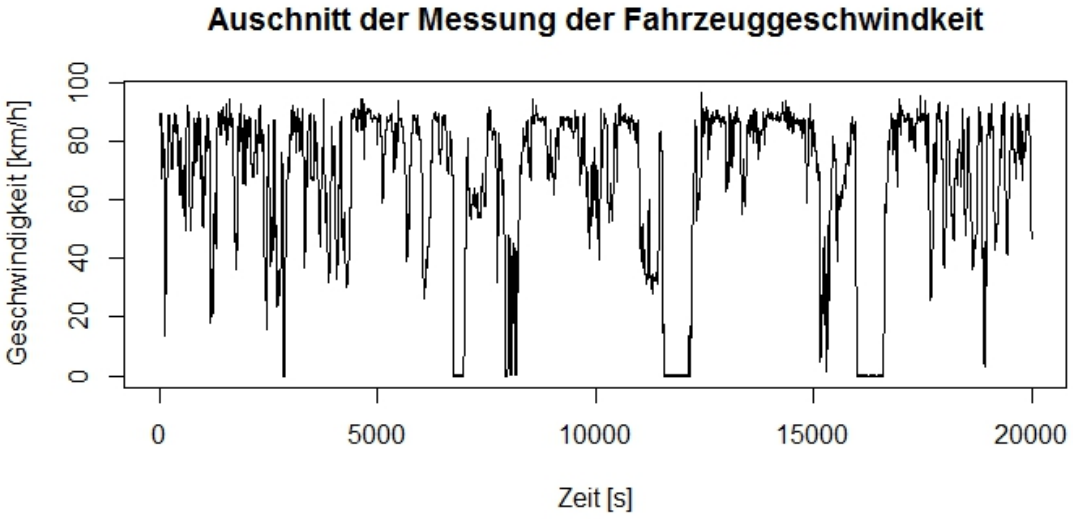


Abbildung 2.6.: Der Auschnitt der Messung der Fahrzeuggeschwindigkeit in dem nach Mustern gesucht wurde.

Es wurden mit allen verwendeten Distanzmaßen mehrere Parameterkombination getestet, wobei sich in fast allen Fällen zeigte, dass entweder so gut wie keine Subwindows oder aber fast alle Subwindows entfernt wurden. In Anhang A.3 sind für jedes Distanzmaß gefundene passende Parameter und einige damit gefundene Muster dargestellt.

2.5. Fazit

Obwohl der Algorithmus auf künstlichen und einfachen realen Daten brauchbare Ergebnisse erzielt, lässt die Performance auf realen Daten stark nach; wohl hauptsächlich, weil wiederkehrende Muster zu selten vorkommen. Damit ist der Algorithmus für das gewünschte Einsatzgebiet aber nur sehr bedingt geeignet.

Die Anzahl und Art der gefundenen Muster hängt stark von den gewählten Parametern und dem Startwert für den Zufallsgenerator ab. Zweitere Abhängigkeit lässt sich vermindern, indem man die Anzahl der Fenster in der Kandidatenmenge erhöht, was aber sehr stark zu Lasten der Rechenzeit geht.

Ein weiteres Problem stellt das Wiederfinden der erkannten Muster dar, welches zur Beschreibung der Zeitreihe durch die Muster unbedingt notwendig ist. Es wurde kurz versucht den vorgestellten Algorithmus dahingehen zu adaptieren (siehe nächster Unterabschnitt), was aber nicht gelang.

2.6. Weitere Überlegungen zum Algorithmus

Hat der Algorithmus dann mehrere Musterinstanzen oder Teile davon gefunden, gilt es nun die „Prototypen“ der vorkommenden Muster zu identifizieren. Dafür muss man die gefundenen Muster zunächst clustern; jeder Cluster enthält dann im Idealfall Teile von

einzelnen Realisierungen des jeweils selben Prototypen. Diese Teile müssen nun passend ausgerichtet werden. Das kann zum Beispiel mithilfe von DTW unter Verwendung des „rigid“ Schrittmusters passieren. Der punktweise Mittelwert der sich überlagernden Teile sollte dann einen Schätzer für den Musterprototypen ergeben.

Als nächstes möchte man nun Instanzen dieser Musterprototypen in der Zeitreihe wiederfinden. Dazu wurde versucht, ähnlich wie beim Entdecken der Muster vorzugehen:

Für ein gegebenes Muster der Länge m unterteilen wir die zur Mustererkennung verwendete Messung in Teile der Länge m, die sich in $m-1$ Punkten überlappen (sliding window). Dann berechnen wir die Ähnlichkeit zwischen dem Muster und jedem Teilstück und vergleichen diesen Wert wieder mit der Ähnlichkeit des Musters zu einem Noise-Window, das gleich erzeugt wird wie im Algorithmus zum Finden der Muster.

Leider zeigte sich, dass diese Methode zum Wiederfinden der Muster nicht geeignet war: Fast immer liegt der Ähnlichkeitswert zwischen Muster und gerade betrachtetem Teilstück unter/über dem mit dem Noise-Window bestimmten Grenzwert und das Muster wird in fast jedem Teilstück „erkannt“. Dies passierte unabhängig vom verwendeten Distanzmaß.

3. Die „bag-of-patterns“ Repräsentation

Als zweite Methode zur Beschreibung der Messungen wurde die „Bag-of-patterns“-Repräsentation (BOP) gewählt. Dieser Ansatz stammt ursprünglich aus dem Gebiet der automatischen Klassifikation von Textdokumenten [Salton et al., 1975]. Das dort verwendete „Vector Space Model“ (siehe Abschnitt 3.1), informell auch „bag-of-words“ genannt, wurde von Lin und Li [2009] für die Verwendung mit Zeitreihen vorgeschlagen und dahingehend adaptiert.

3.1. Das Vector Space Model in der Klassifikation von Textdokumenten

Gegeben ist hier eine Menge von Textdokumenten die geclustert, oder bestehenden Clustern zugeordnet werden sollen. Im Vector Space Model wird dazu jedes Dokument als Vektor im $\mathbb{R}^n$ repräsentiert. Die Dimension n entspricht dabei der Anzahl der unterschiedlichen Wörter die in der Gesamtheit der Textdokumente vorkommen. Jede Komponente des Vektors ist einem bestimmten Wort zugeordnet und ihr (ganzzahliger) Wert zählt das Vorkommen dieses Wortes im aktuellen Dokument. Gegebenenfalls wird der Vektor dann noch normalisiert um unterschiedliche Textlängen auszugleichen. Das Resultat ist dann der „bag-of-words“, also so etwas wie ein „Histogramm“ der vorkommenden Wörter.

Man bemerke, dass bei dieser Art der Repräsentation jede Art von Reihenfolge verlorengeht, es kommt nur darauf an, wie oft die verschiedenen Wörter auftreten.

Will man nun für Zeitreihen ähnlich vorgehen, so ist das Prinzip klar: Die Wörter im Textdokument entsprechen Mustern in der Zeitreihe und wir zählen wie oft welches Muster auftritt. Dabei stellen sich zunächst aber zwei Probleme: Einerseits ist nicht klar, was die Menge der unterschiedlichen Wörter bzw. Muster sein soll, und wie deren Vorkommen gezählt wird; zwei exakt gleiche Realisierungen des gleichen Musters werden in der Zeitreihe nicht vorkommen. Andererseits fehlt die klare Abgrenzung von einem Muster zum nächsten, die bei Wörtern kein Problem darstellt.

Lin und Li [2009] schlagen zur Lösung dieser Probleme die von Lin et al. [2007] entwickelte SAX-Repräsentation von Zeitreihen vor, die im nächsten Abschnitt vorgestellt wird.

3.2. Symbolic Aggregate Approximation (SAX)

Die SAX-Repräsentation stellt eine Zeitreihe der Länge n als ein Wort von fixer Länge w über einem Alphabet mit (endlicher) Mächtigkeit a dar. Wir betrachten hier den Spezialfall in dem $w \mid n$ hält, das heißt $\exists z \in \mathbb{N} : w \cdot z = n$. Die SAX-Repräsentation ist aber leicht auf den allgemeinen Fall generalisierbar [Lin et al., 2007], der allerdings in dieser Arbeit nicht benötigt wird. Des weiteren habe die Zeitreihe das arithmetische Mittel 0 und die Standardabweichung 1.

Gegeben sei nun eine Zeitreihe $C = c_1, \ldots, c_n$ mit $\text{mean}(C) = 0$ und $\text{sd}(C) = 1$, von der wir die SAX-Repräsentation berechnen wollen, sowie die Wortlänge w und das Alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_a\}$.

Der erste Schritt besteht in der Berechnung der „Piecewise Aggregate Approximation“ (PAA) der Ordnung w. Dazu wird die Zeitreihe in w gleichgroße Stücke geteilt ($w \mid n$ nach Voraussetzung) und jedes Stück durch seinen Mittelwert repräsentiert. Wir erhalten also die PAA der Zeitreihe als $\bar{C} = \bar{c}_1, \ldots, \bar{c}_w$, mit

$$\bar{c}_i = \frac{w}{n} \sum_{j=\frac{n}{w}(i-1)+1}^{\frac{n}{w}i} c_j, \qquad \forall i = 1, \ldots, w.$$

Im zweiten Schritt wird der Wertebereich der Zeitreihe in a Stücke geteilt, und zwar so, dass alle Stücke unter Standard-Normalverteilung die gleiche Wahrscheinlichkeit aufweisen. Formal definieren wir also $a+1$ Breakpoints $\beta_0, \ldots, \beta_a \in \mathbb{R} \cup \{-\infty, \infty\}$ wobei für $j = 0, \ldots, a$ die $\beta_j = F^{-1}_{N(0,1)}(\frac{j}{a})$ die passenden Quantile der Standard-Normalverteilung sind. Jedem der a Stücke wird dann ein Symbol aus Σ zugewiesen.

Die SAX-Repräsentation der Zeitreihe $\hat{C} = \hat{c}_1, \ldots, \hat{c}_w$ ergibt sich dann als das Wort jener Symbole, in deren Bereichen die entsprechenden PAA-Mittelwerte liegen, also

$$\hat{c}_i = \sigma_j \iff \beta_{j-1} \leq \bar{c}_i < \beta_j, \qquad \forall i = 1, \ldots, w.$$

Das Prinzip wird in Abbildung 3.1 noch einmal verdeutlicht.

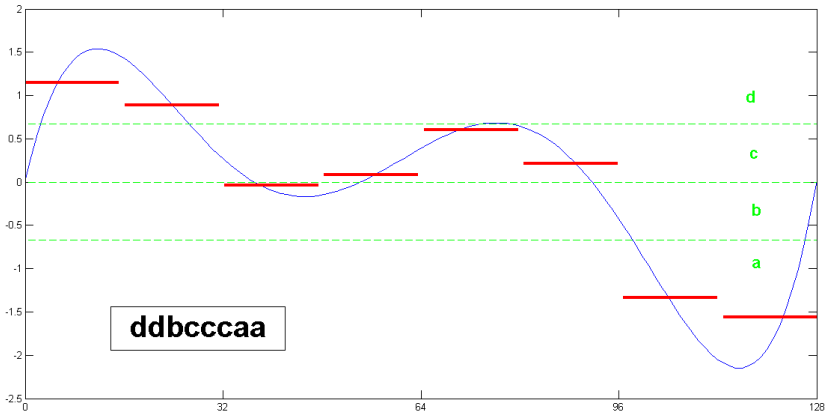


Abbildung 3.1.: Beispiel einer SAX-Repräsentation (schwarz) mit $w = 8$ und $\Sigma = \{a, b, c, d\}$. In rot eingezeichnet ist die PAA-Repräsentation der Zeitreihe, die drei grün-strichlierten Linien stellen, zusammen mit $\{-\infty, \infty\}$ die fünf Breakpoints dar.

3.3. Erzeugung des Feature-Vektors

Nun wollen wir die SAX-Repräsentation nutzen, um die Anzahl der verschiedenen Muster/Wörter in einer langen Zeitreihe zu bestimmen. Dazu legen wir zunächst alle für SAX

benötigten Parameter fest, und wählen zusätzlich eine Fensterlänge l, wobei $w \mid l$ gelten soll. Zur Erzeugung des Feature-Vektors (entspricht dem „Worthistogramm“ aus der „bag-of-words“ Methode) der die Zeitreihe dann repräsentieren soll, gehen wir wie folgt vor:

Jede Komponente des Vektors soll einem SAX-Wort entsprechen, das heißt der Vektor ist von der Dimension a^w. Zunächst sind alle Koordinaten des Vektors 0. Der Vektor selbst wird befüllt, indem wir ein sliding-window mit Breite l über die Zeitreihe ziehen. Das Stück der Zeitreihe das im jeweils aktuellen Fenster liegt wird bezüglich Mittelwert und Standardabweichung normalisiert und in das korrespondierende SAX-Wort übergeführt. Die dem Wort entsprechende Koordinate des Vektors erhöhen wir dann um 1. Ergeben mehrere direkt aufeinander folgende Fenster das gleiche SAX-Wort, so wird dieses, den Autoren der Methode folgend, jedoch nur einmal gezählt. Zum Schluss wird der Vektor noch auf die Koordinatensumme 1 normiert, womit der Wert einer Koordinate in etwa mit der Auftrittswahrscheinlichkeit des entsprechenden Wortes korrespondiert.

Das gesamte Vorgehen soll nun an einem Beispiel demonstriert werden:

Beispiel. [Erzeugung eines SAX Feature-Vektors] Es sei eine Zeitreihe der Länge 8 gegeben, und zwar $C = 2, 2, 1, -2, -3, -2, -2, 3$. Wir wählen eine Fensterlänge $l = 6$, eine SAX-Wortlänge von $w = 3$ und das Alphabet $\Sigma = \{a, b\}$. Damit sind die Breakpoints $\{-\infty, 0, \infty\}$ und der Feature-Vektor hat die Länge $2^3 = 8$. Wir schieben nun das sliding window über die Zeitreihe und berechnen für jedes Fenster das zugehörige SAX-Wort (Abbildung 3.2).

Die gefundenen Worte werden nun im Feature-Vektor eingetragen (wobei *baa* nur einmal gezählt wird, weil beide Vorkommen direkt hintereinander liegen). Der nicht normalisierte Vektor ergibt sich also als:

aaa	*aab*	*aba*	*abb*	*baa*	*bab*	*bba*	*bbb*
(0	0	0	0	1	1	0	0)

Dieser muss nun abschließend noch durch die Summe seiner Einträge, also durch 2, dividiert werden.

◊

3.4. Erste Beobachtungen

Für erste Tests mit dieser Methode wurden Messungen der Geschwindigkeit von LKWs aus mehreren Anwendungsszenarien verwendet. Es waren gegeben:

- 13 Messungen von Fahrten auf Autobahn/Bundesstraße
- 14 Messungen von Fahrten auf einer Passstrecke
- 13 Messungen von Müllfahrzeugen
- 9 Messungen von Fahrten in der Stadt
- 6 Messungen von Verteilerverkehr

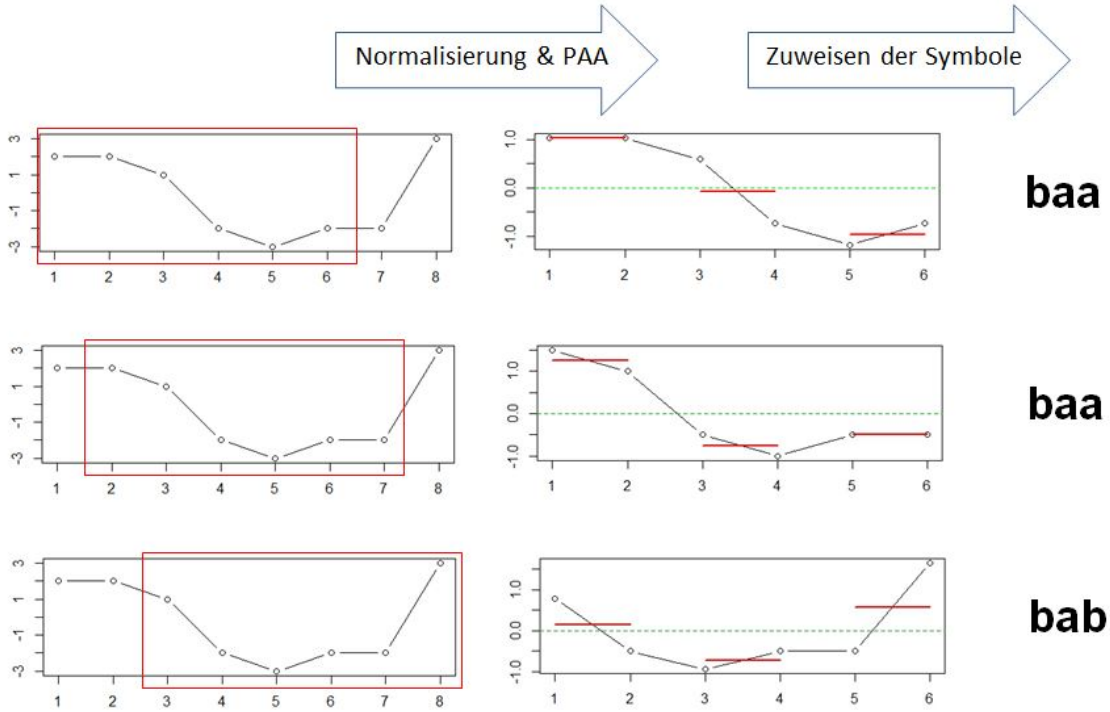


Abbildung 3.2.: Linke Spalte: ein sliding window wird über die Zeitreihe gezogen. Mittlere Spalte: Das aktuelle Fenster wird normiert und die PAA Segmente berechnet (rot). Rechte Spalte: Die Wörter ergeben sich aus der Lage der PAA-Segmente (rot) in Bezug zu den Breakpoints (grün).

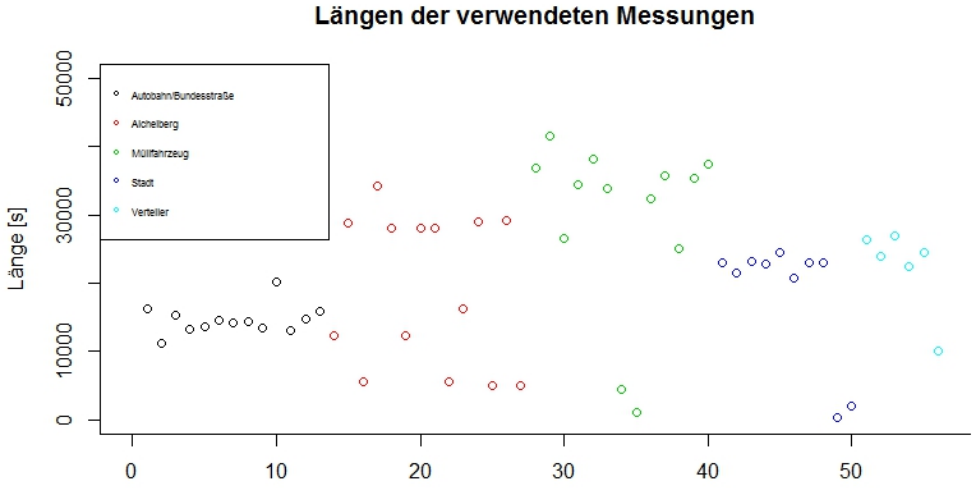


Abbildung 3.3.: Übersicht der Längen der verwendeten Messungen, die Auflösung betrug 1Hz.

Die Längen dieser 55 Messungen sind in Abbildung 3.3 angeführt.

Für diese Messungen wurden nun die Feature-Vektoren des Geschwindigkeitskanals nach der BOP-Methode berechnet. Die dabei verwendeten Parameter waren eine Fensterlänge von $l = 96$, eine Wortlänge von $w = 6$ und ein Alphabet mit $a = 4$ Buchstaben. Das ergibt eine Länge von 4096 für die Feature-Vektoren. Beim Betrachten der erzeugten Feature-Vektoren zeigt sich, dass die meisten Wörter gar nicht vorkommen, also sehr viele Koordinaten, nämlich 1391 von 4096 in allen Gruppen 0 sind. Diese können natürlich sofort ohne Informationsverlust entfernt werden. In Abbildung 3.4 sind die Feature-Vektoren der jeweils selben Gruppen koordinatenweise gemittelt dargestellt. Beispiele der Feature-Vektoren von verschiedenen Gruppen finden sich in Anhang B. Schon hier ist ersichtlich, dass die Vektoren in der gleichen Gruppe relativ ähnlich sind, während es zwischen den verschiedenen Gruppen größere Unterschiede gibt.

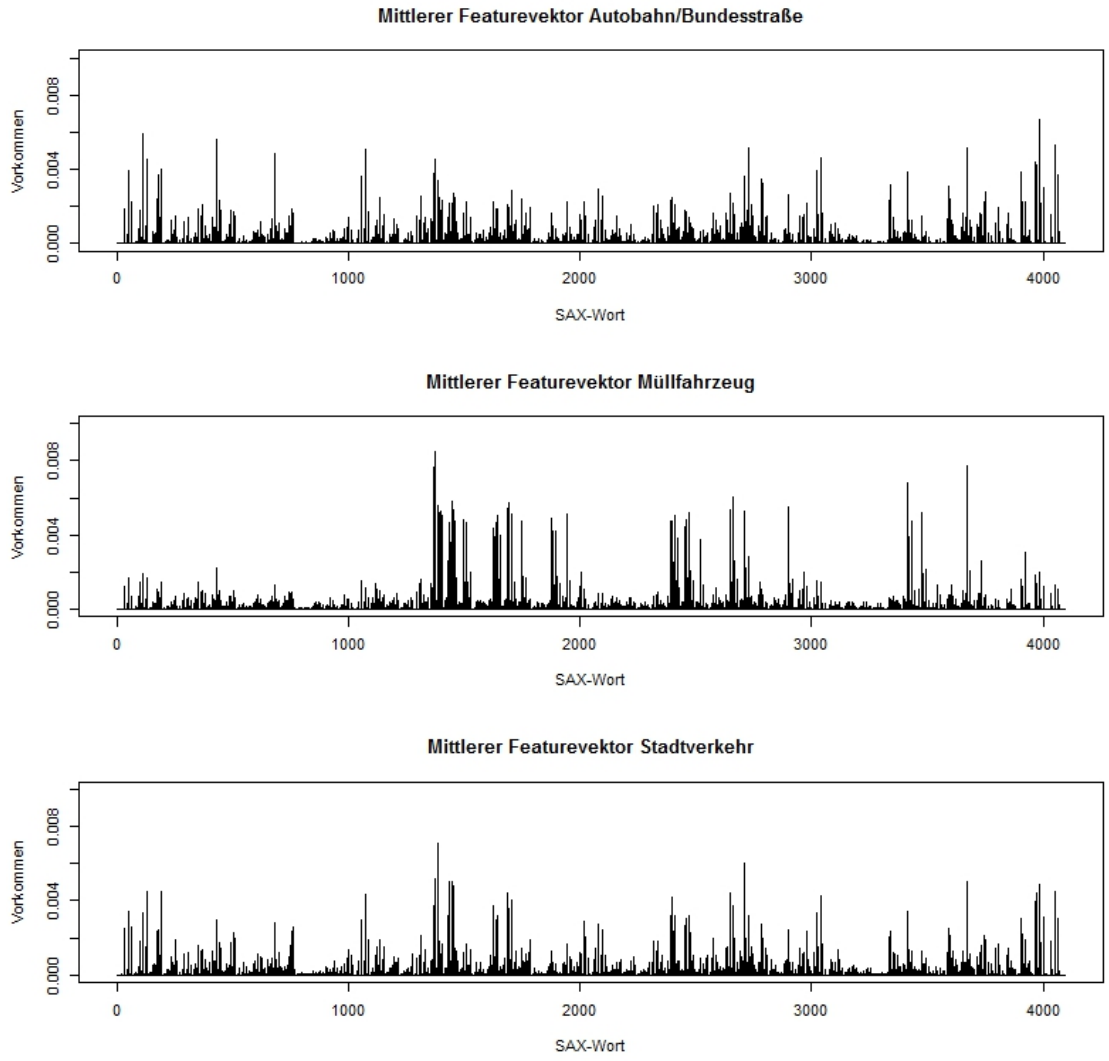


Abbildung 3.4.: Koordinatenweises arithmetisches Mittel der Feature-Vektoren von verschiedenen Gruppen

Es stellte sich heraus, dass mit dieser Methode Daten die sehr unterschiedlich sind, wie etwa Müllfahrzeug- und Autobahndaten, schon mit einfachen Mitteln gut diskriminiert werden können. Wendet man etwa agglomeratives, hierarchisches, complete-linkage Clustern mit Manhattan-Metrik auf die insgesamt 26 Feature-Vektoren dieser beiden Gruppen an, erhält man das Dendrogramm in Abbildung 3.5, in dem die beiden Gruppen schön durch große Höhe getrennt sind und nur eine Beobachtung als zu keiner Gruppe passend auffällt. Eine genaue Beschreibung dieser Clustermethode und der damit erzeugten Dendrogramme befindet sich in Abschnitt 6.1.

Wendet man diese Cluster-Methode jedoch auf Gruppen an die nicht so verschieden wie Müllfahrzeug- und Autobahndaten sind, so erhält man weniger gute, wenn auch nicht wirklich schlechte Ergebnisse. Als Beispiel sei das Dendrogramm in Abbildung 3.6 gezeigt, in dem das Clusterergebnis von Daten aus den Gruppen „Passstrecke“ (also eine besondere Autobahnstrecke) und „Autobahn / Bundesstraße“ dargestellt ist.

3.5. Adaption der „bag-of-pattern“-Methode

Nun wurde versucht die „klassische“ Methode zum Erzeugen der SAX-Wörter an unsere Gegebenheiten anzupassen, um nach Möglichkeit alle Daten gut clustern zu können und nicht nur die Müllfahrzeuge abzutrennen.

3.5.1. Aufgabe der Normalisierung

Als erste Adaption wurde die Normalisierung der Daten im sliding window aufgegeben. Diese Normalisierung ist zwar in fast allen Anwendungen Standard [Keogh und Kasetty, 2003], könnte aber in unserem Setting zu unerwünschten Ergebnissen führen. So macht es natürlich einen großen Unterschied ob sich ein Fahrzeug konstant mit 100 km/h oder konstant mit 15 km/h bewegt. Mit Normalisierung würde jedoch beides zum gleichen SAX-Wort führen.

Dadurch ergibt sich aber die Frage wie denn nun die Breakpoints zu wählen sind. Eine Möglichkeit wäre natürlich die Verwendung der empirischen Quantile der Daten, wir haben uns aber entschieden zunächst fest gewählte, äquidistante Breakpoints zu versuchen. Dies waren für die Fahrzeuggeschwindigkeit die Werte 25 km/h, 50 km/h und 75 km/h.

3.5.2. Gewichtung der Komponenten

Zusätzlich zur Aufgabe der Normalisierung wurden von nun an die Komponenten des Feature-Vektoren gewichtet. Die dazu verwendete Methode stammt aus dem Gebiet des Text Mining / Information Retrieval und heißt „term frequency–inverse document frequency“, kurz tf-idf. Tf-idf ist eine Statistik die für jedes Wort in einem Textdokument berechnet wird und umso größer ist, je besser das Wort das Dokument beschreibt [Rajaraman und Ullman, 2011]. Tf-idf wird dabei immer größer je öfter ein Wort im Dokument vorkommt und immer kleiner je öfter ein Wort in der Gesamtheit der Dokumente vorkommt.

Für die Definition der tf-idf Statistik werden wir die folgende Notation verwenden:

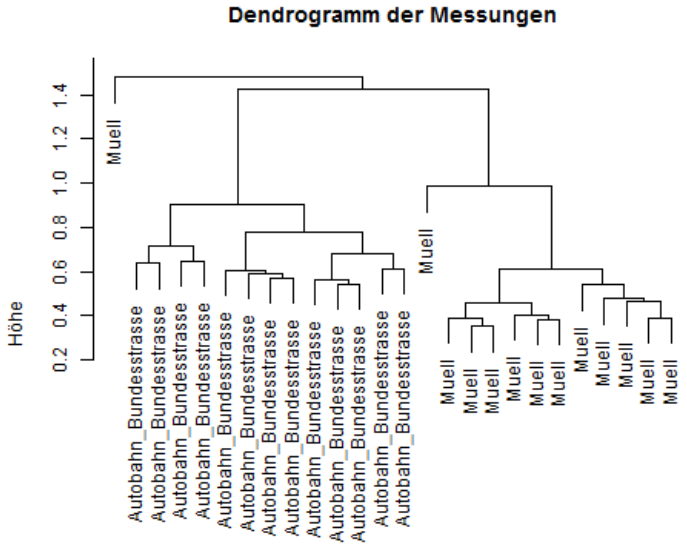


Abbildung 3.5.: Cluster-Dendrogramm von Müllfahrzeug- und Autobahndaten. Bis auf die eine untypische Beobachtung gibt es keine falsch zugeordneten Beobachtungen.

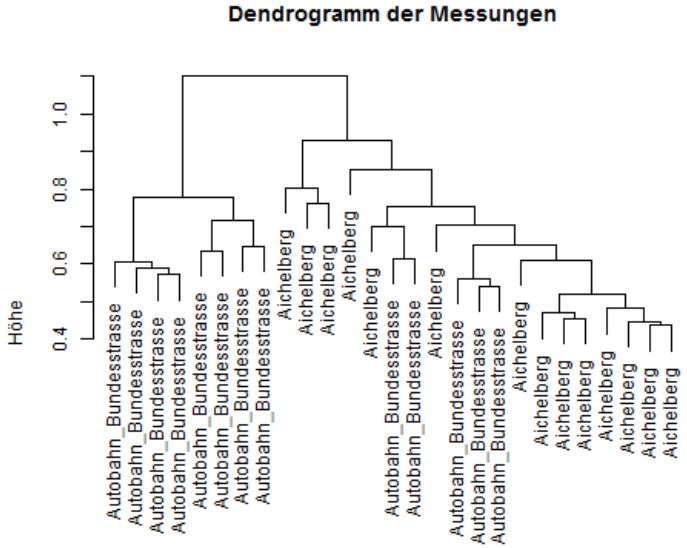


Abbildung 3.6.: Cluster-Dendrogramm von Daten aus den Gruppen „Passstrecke“ und „Autobahn / Bundesstraße“. Bei der Einteilung in 2 Cluster würden 5 Messungen der Gruppe „Autobahn / Bundesstrasse“ falsch zugeordnet werden.

Symbol	Bedeutung
M	Menge der Messungen / Zeitreihen
$m\ (\in M)$	einzelne Messung (aus der Menge der Messungen)
$s\ (\in m)$	einzelnes SAX-Wort (in einer Messung)
$h(s, m)$	Anzahl der Vorkommen von s in m (gezählt wie bei der Erzeugung der Feature-Vektoren)

Die tf-idf Statistik ergibt sich als Produkt der zwei namensgebenden Teile:

$$tfidf(s, m, M) = tf(s, m) \cdot idf(s, M).$$

Dabei misst der erste Teil die Vorkommenshäufigkeit eines Wortes im Dokument. Für die genaue Definition gibt es verschiedene Ansätze [Manning et al., 2008], etwa die binäre „Häufigkeit“ ($1 \triangleq$ kommt vor, $0 \triangleq$ kommt nicht vor), die „normale“ relative Häufigkeit, oder dieselbe logarithmisch skaliert. Wir haben uns schließlich für eine vierte Variante entschieden, die unterschiedliche Längen der Dokumente (hier also Messungen) besonders gut ausgleichen soll und wie folgt definiert ist:

$$tf(s, m) = \frac{1}{2} + \frac{\frac{1}{2} \cdot h(s, m)}{\max_s h(s, m)}.$$

Der zweite Teil der Statistik soll messen, wie oft (eigentlich wie selten) ein (SAX-)Wort in der Gesamtheit der Dokumente (Messungen) vorkommt. Zu diesem Zweck bestimmen wir die relative Häufigkeit der Anzahl der Dokumente, in denen das Wort überhaupt vorkommt, und nehmen den Kehrwert um statt der Häufigkeit die „Seltenheit“ zu messen. Das Ganze wird dann noch aus informationstheoretischen Gründen logarithmiert. Wir haben also:

$$idf(s, M) = \log \frac{|M|}{|\{m \in M : s \in m\}|}.$$

Wir bemerken, dass die verwendete Basis des Logarithmus für die Bewertung keine Rolle spielt, da sich Logarithmen zu verschiedenen Basen ja nur um eine multiplikative Konstante unterscheiden.

Wenn wir die tf-idf Statistik betrachten, erkennen wir, dass der tf-Teil (als relative Häufigkeit) ja eigentlich genau den Koordinaten unserer Feature-Vektoren entspricht. Multiplizieren wir diese also komponentenweise mit dem entsprechenden idf-Teil, so haben wir genau die tf-idf Statistik. Wir können diese also als Gewichtung unserer Feature-Vektoren interpretieren.

Nachdem nun verschiedene Adaptionen der ursprünglichen BOP-Methode vorgestellt wurden, sollen diese auch getestet und evaluiert werden. Dies passiert im folgenden Kapitel.

4. Performance des BOP-Ansatzes mit verschiedenen Parametern

Weil die Repräsentation der Messungen als bag-of-patterns in ersten Tests vielversprechende Ergebnisse gezeigt hatte, wurde die Methode und der Einfluss der SAX-Parameter nun genauer untersucht. Zu diesem Zweck wurde eine Support-Vector-Machine -kurz SVM- trainiert, und deren Klassifizierungsfehler bei der Zuordnung der Messungen zu den einzelnen Klassen betrachtet. Bevor die Ergebnisse präsentiert werden, soll hier jedoch zuerst das Konzept und die Funktionsweise einer SVM noch einmal erläutert werden.

4.1. Die Support-Vector-Machine

Eine Support-Vector-Machine ist ein Verfahren zur Mustererkennung, genauer zur (zunächst) binären Klassifizierung. Das heißt, das Verfahren erhält als Input mehrere beobachtete Merkmale einer Messung in Form von Feature-Vektoren und liefert als Output zu welcher der zwei möglichen Klassen die Messung (wahrscheinlich) gehört. Bevor die SVM auf diese Weise verwendet werden kann, muss sie zunächst *trainiert* werden. Für das Training erhält die SVM eine Menge von Feature-Vektoren und zusätzlich die Information zu welcher der beiden Klassen diese jeweils gehören. Dabei werden die Feature-Vektoren als Punkte im Raum betrachtet. Über die Lösung eines quadratischen Optimierungsproblems wird dann eine Hyperebene gesucht welche die Punkte der beiden Klassen trennt. Zusätzlich soll diese Hyperebene zu den am nächsten liegenden Punkten der beiden Klassen maximalen Abstand haben, das heißt die SVM ist ein *large margin classifier*.

Formal haben wir also folgendes:

Gegeben sind m Tupel $(\vec{x}_i, y_i)$, wobei $\vec{x}_i \in \mathbb{R}^n$ die Feature-Vektoren, und $y_i \in \{-1, 1\}$ die zugehörigen Klassen sind. Die trennende Hyperebene habe die Form $\langle \vec{w}, \vec{x} \rangle + b = 0$, wobei $\langle \cdot, \cdot \rangle$ das Standard-Skalarprodukt im $\mathbb{R}^n$ ist. Der Gewichtsvektor $\vec{w}$ beschreibt die Richtung der Hyperebene, während b ihren Abstand vom Ursprung bestimmt. Die skalare Größe b wird in diesem Zusammenhang auch als *bias* bezeichnet.

Die *Entscheidungsfunktion*, die bestimmt zu welcher Klasse ein Feature-Vektor gehört, ist dann $f(\vec{x}) = \text{sign}(\langle \vec{w}, \vec{x} \rangle + b)$.

Man kann zeigen, dass für SVMs der *Generalisierungsfehler* nach oben beschränkt ist. Der Generalisierungsfehler ist dabei der erwartete Fehler den die SVM beim Klassifizieren von neuen (also nicht zum Trainieren verwendeten) Instanzen macht. Diese Fehlerschranke hat unter anderem zwei wichtige Eigenschaften [Vapnik, 1998]:

1. Die Fehlerschranke wird minimiert, in dem der Abstand der trennenden Hyperbebene zu den Punkten der beiden Klassen maximiert wird.

2. Die Fehlerschranke hängt nicht von der Dimension der Feature-Vektoren ab.

Wir möchten nun die trennende Hyperebene natürlich so wählen, dass Eigenschaft 1 erfüllt ist, also der Abstand zu den Punkten der beiden Klassen maximiert wird. Es ist klar, dass sofern irgendeine trennende Hyperebene existiert, die Feature-Vektoren also *linear separierbar* sind, auch (zumindest) eine Hyperebene mit maximalen Abstand zu den beiden Punktwolken existiert. Um diese zu finden, also die SVM zu trainieren, gehen wir wie folgt vor:

Seien $(\vec{x}_1, 1)$ und $(\vec{x}_2, -1)$ zwei Instanzen aus einer Menge von Trainingsdaten, so dass $\vec{x}_1$ und $\vec{x}_2$ minimale Distanz unter allen Distanzen von Punkten aus verschiedenen Klassen haben. Das Argument der sign-Funktion in der Entscheidungsfunktion ist natürlich invariant gegenüber positiven Skalierungen. Wir können also o.B.d.A. $\vec{w}$ und b so skalieren, dass $\langle \vec{w}, \vec{x}_1 \rangle + b = 1$ und $\langle \vec{w}, \vec{x}_2 \rangle + b = -1$ hält. Diese beiden Gleichungen beschreiben zwei zur optimalen Hyperebene kollinare Ebenen, die durch $\vec{x}_1$ bzw. $\vec{x}_2$ verlaufen. Zwischen diesen beiden *kanonischen Hyperebenen* liegt ein Bereich in dem sich die optimale Hyperebene befindet, in dem aber keine Feature-Vektoren liegen.

Der Abstand der optimalen Hyperebene zu den kanonischen Hyperebenen ergibt sich also durch die Länge der Projektion der Strecke $[\vec{x}_1, \vec{x}_2]$ auf den Einheitsnormalvektor der Hyperebene, also als

$$\frac{\langle (\vec{x}_1 - \vec{x}_2), \vec{w} \rangle}{\|\vec{w}\|}.$$

Aus den Gleichungen der kanonischen Hyperebenen können wir aber zusätzlich $\langle (\vec{x}_1 - \vec{x}_2), \vec{w} \rangle = 2$ folgern, womit sich der Abstand zu $\frac{2}{\|\vec{w}\|}$ vereinfacht. Wollen wir diesen Ausdruck maximieren, so ist das äquivalent zur Minimierung von $\frac{1}{2}\|\vec{w}\|^2$. Alleine durch diese Minimierung können wir $\vec{w}$ aber natürlich noch nicht berechnen, wir müssen zusätzlich die korrekte Klassifizierung der Trainingsdaten fordern!

Insgesamt erhalten wir also die optimale trennende Hyperebene durch Lösen des folgenden quadratischen Programmes:

$$(S) \begin{cases} \text{minimiere} & \frac{1}{2}\|\vec{w}\|^2 \\ \text{s.t.:} & y_i(\langle \vec{w}, \vec{x}_i \rangle + b) \geq 1, \qquad i = 1, \ldots, m. \end{cases}$$

Dieses Programm können wir lösen, indem wir die assoziierte Lagrange-Funktion minimieren, das heißt

$$\min_{\vec{w}, b, \alpha_i \geq 0} L(\vec{w}, b, \vec{\alpha}) = \min_{\vec{w}, b, \alpha_i \geq 0} \frac{1}{2}\langle \vec{w}, \vec{w} \rangle - \sum_{i=1}^{m} \alpha_i (y_i(\langle \vec{w}, \vec{x}_i \rangle + b) - 1). \tag{4.1}$$

Im Minimum gilt:

$$\frac{\partial L}{\partial \vec{w}} = \vec{w} - \sum_{i=1}^{m} \alpha_i y_i \vec{x}_i = 0, \tag{4.2}$$

$$\frac{\partial L}{\partial b} = -\sum_{i=1}^{m} \alpha_i y_i = 0. \tag{4.3}$$

Berechnen wir daraus $\vec{w}$, setzen in (4.1) ein, und vereinfachen, so erhalten wir die duale Formulierung nach Wolfe:

$$W(\vec{\alpha}) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \langle \vec{x}_i, \vec{x}_j \rangle. \tag{4.4}$$

Um in dieser Formulierung die optimale Lösung zu finden, müssen wir folgendes quadratisches Programm lösen:

$$(W) \begin{cases} \text{maximiere} & W(\vec{\alpha}) \\ \text{s.t.:} & \sum_{i=1}^{m} \alpha_i y_i = 0 \\ & \alpha_i \geq 0, \qquad i = 1, \ldots, m. \end{cases}$$

4.1.1. Der Kernel-Trick

Wir bemerken, dass in der Definition von $W(\vec{\alpha})$ die Feature-Vektoren $\vec{x}_i$ nur in einem Skalarprodukt vorkommen. Zusammen mit der Eigenschaft der SVM, dass die Fehlerschranke nicht von der Dimension des Raumes abhängt, lässt das den *Kernel-Trick* zu, der wie folgt funktioniert:

In vielen Fällen werden die Feature-Vektoren nicht linear separierbar sein. Deshalb möchten wir diese Vektoren in einen größeren (eventuell sogar unendlichdimensionalen) Raum einbetten, in dem sie dann linear separierbar sein sollen. Die oben angesprochene Vorgehensweise ändert sich dadurch nicht, nur das Skalarprodukt in (4.4) ist dann das des neuen, größeren Raumes. Eine passende Einbettung zu finden ist aber im allgemeinen sehr rechenintensiv und kann nicht leicht durchgeführt werden. Gibt es jedoch einen Hilbertraum $\mathfrak{H}$, eine Einbettung $\Phi : \mathbb{R}^n \to \mathfrak{H}$ und eine Funktion $K : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ so dass

$$K(\vec{x}_i, \vec{x}_j) = \langle \Phi(\vec{x}_i), \Phi(\vec{x}_j) \rangle_{\mathfrak{H}}, \qquad \forall \vec{x}_i, \vec{x}_j$$

hält, dann muss die Einbettung Φ nicht explizit bekannt sein, es reicht die Kenntnis des Kernels K. Die Formulierung von W ändert sich nur durch das Ersetzen des Skalarproduktes durch die allgemeinere Kernelfunktion:

$$W'(\vec{\alpha}) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j K(\vec{x}_i, \vec{x}_j).$$

Natürlich ist hier die ursprüngliche Formulierung als Spezialfall durch Wahl des *linearen Kernels* $K(\vec{x}_i, \vec{x}_j) = \langle \vec{x}_i, \vec{x}_j \rangle$ enthalten. Ein anderer oft verwendeter Kernel ist der *Gauss-Kernel*, gegeben durch

$$K_\sigma(\vec{x}_i, \vec{x}_j) = \exp\left(-\frac{\|\vec{x}_i - \vec{x}_j\|^2}{2\sigma^2} \right)$$

.

Die Klasse der möglichen Kernelfunktionen ist funktionalanalytisch durch den *Satz von Mercer* charaktersiert, auf den jedoch hier nicht eingegangen werden soll. Wichtig ist, dass wir mit Hilfe des Kernel-Tricks nun auch Klassen trennen können, die nicht linear separierbar sind.

Haben wir eine Optimallösung $\vec{\alpha}^*$ von Problem (W) gefunden, so erhalten wir den zugehörigen optimalen bias b^* aus den Gleichungen der zwei kanonischen Hyperebenen, in denen $\vec{w}$ wie vorhin durch $\vec{y}$ und $\vec{\alpha}$ ausgedrückt wird.

Schlussendlich erhalten wir dann das zugehörige $\vec{w}^*$ aus $(\vec{\alpha}^*, b^*)$ über die partielle Ableitung der Lagrange-Funktion der $\Phi(\vec{x}_i)$. Analog zu (4.2) erhalten wir dort:

$$\vec{w}^* = \sum_{i=1}^{m} \alpha_i^* y_i \Phi(\vec{x}_i).$$

Die verallgemeinerte Entscheidungsfunktion ergibt sich damit als:

$$\begin{aligned} f(\vec{x}) &= \operatorname{sign}\left(\langle \vec{w}^*, \Phi(\vec{x})\rangle_{\mathfrak{H}} + b^*\right) = \\ &= \operatorname{sign}\left(\left\langle \sum_{i=1}^{m} \alpha_i^* y_i \Phi(\vec{x}_i), \Phi(\vec{x}) \right\rangle_{\mathfrak{H}} + b^*\right) = \\ &= \operatorname{sign}\left(\sum_{i=1}^{m} \alpha_i^* y_i K(\vec{x}_i, \vec{x}) + b^*\right). \end{aligned}$$

Nun wollen wir noch die Rolle der α_i etwas näher betrachten. Aus den Karush-Kuhn-Tucker-Bedingungen [Kuhn und Tucker, 1951] folgt, dass $\alpha_i > 0$ nur für jene $\vec{x}_i$ gilt, die der optimalen trennenden Hyperebene am nächsten sind (aktive Nebenbedingungen). Alle anderen α_i sind 0. Damit hängt die Entscheidungsfunktion aber nur von jenen $\vec{x}_i$ ab, die der optimalen trennenden Hyperebene am nächsten liegen, den *Stützvektoren*. Damit erklärt sich auch der Name Support-Vector-Machine. Die α_i messen dabei den Einfluss den eine einzelne Beobachtung $\vec{x}_i$ auf die Lage der Hyperebene hat.

4.1.2. Weitere Überlegungen zur Support-Vector-Machine

Auch unter Verwendung des Kernel-Tricks kann es vorkommen, das Daten nicht separierbar sind (etwa zwei exakt gleiche Feature-Vektoren in verschiedenen Klassen), oder dass Ausreißer einer Klasse sehr großen Einfluss auf die Lage der trennenden Hyperebene haben. Aus diesem Grund möchte man das Optimierungsproblem (S) relaxieren, um auch falsch klassifizierte Trainingsdaten zuzulassen. Dafür fordern wir aber zusätzlich, dass die α_i nach oben beschränkt sind, also der Einfluss von einzelnen Beobachtungen nicht zu groß werden darf. Dazu betrachten wir das folgende, eng mit (S) verwandte, quadratische Optimierungsproblem:

$$(S_{relax}) \begin{cases} \text{minimiere} & \frac{1}{2}\|\vec{w}\|^2 + C\sum_{i=1}^{m} \xi_i \\ \text{s.t.:} & y_i(\langle \vec{w}, \vec{x}_i\rangle + b) \geq 1 - \xi_i, \qquad i = 1, \ldots, m \\ & \xi_i \geq 0, \qquad i = 1, \ldots, m. \end{cases}$$

Wir haben also zu (S) nicht negative Schlupfvariablen hinzugefügt, die es möglich machen, dass Daten falsch klassifiziert werden. Die Summe dieser Schlupfvariablen kommt jedoch auch in der Zielfunktion, gewichtet mit einem Faktor C vor, sodass nicht zu viele Daten falsch klassifiziert werden können. Wir betrachten nun die Lagrange-Funktion des relaxierten Problems und ihre partiellen Ableitungen im Minimum:

$$L(\vec{w}, b, \vec{\alpha}, \vec{\beta}) = \frac{1}{2}\langle \vec{w}, \vec{w} \rangle + C \sum_{i=1}^{m} \xi_i - \sum_{i=1}^{m} \alpha_i (y_i (\langle \vec{w}, \vec{x}_i \rangle + b) - 1 + \xi_i) - \sum_{i=1}^{m} \beta_i \xi_i$$

$$\frac{\partial L}{\partial \vec{w}} = \vec{w} - \sum_{i=1}^{m} \alpha_i y_i \vec{x}_i = 0 \tag{4.5}$$

$$\frac{\partial L}{\partial \vec{b}} = -\sum_{i=1}^{m} \alpha_i y_i = 0 \tag{4.6}$$

$$\frac{\partial L}{\partial \xi_i} = C - \alpha_i - \beta_i = 0. \tag{4.7}$$

Wir bemerken, dass wir unter Verwendung von (4.5) und (4.6) genau die gleiche duale Zielfunktion wie für (S) erhalten. Wegen (4.7) gilt jedoch zusätzlich, dass $\alpha_i \leq C$ ist, denn die β_i sind als Lagrange-Multiplikatoren nicht negativ. Auch hier kann man analog zu vorhin den Kernel-Trick verwenden. Diese Klassifizierungsmethode heißt —wegen der oberen Schranke C für die α_i— *C-Classification*.

Als letzter Punkt bleibt noch die Behandlung von mehreren Klassen zu klären, denn bis jetzt konnten ja immer nur zwei Klassen unterschieden werden. Dafür werden jeweils mehrere SVMs trainiert:

Bei *one-vs-all* Strategien wird für jede Klasse eine SVM trainiert, die jeweils entscheiden soll, ob eine Beobachtung aus eben dieser Klasse kommt oder nicht. Ergeben mehrere SVMs, dass eine Beobachtung zu „ihrer“ Klasse gehört, so „gewinnt“ diejenige bei der in der Entscheidungsfunktion im Signum das größere Argument steht. Dieses Argument ist ja ein Maß dafür, wie weit die Beobachtung von der trennenden Hyperebene entfernt ist, und tendenziell gehören weiter entfernte Punkte sicherer zu ihrer jeweiligen Klasse.

Bei *one-vs-one* Strategien werden SVMs trainiert, die jeweils zwei Klassen direkt miteinander vergleichen. Deren Entscheidungen werden dann in einer vorher festgelegten Reihenfolge abgefragt, bis feststeht aus welcher Klasse die Beobachtung stammt.

Schlussendlich kann man auch noch *one-class-classifier* verwenden. Dabei geht es eigentlich darum, Änderungen in neu hinzukommenden Daten zu entdecken (*novelty-detection*). Dazu wird eine SVM mit nur einer Klasse trainiert; deren Entscheidungsfunktion ist dann im „Inneren“ der Daten positiv und außerhalb negativ. Trainiert man nun für jede Klasse einen one-class-classifier, kann man die Entfernungen der neuen Daten zu den Entscheidungsgrenzen der jeweiligen Klassen messen und kann so die Klassenzugehörigkeit bestimmen.

4.1.3. Validierung

Für die Bewertung einer trainierten SVM verwendet man üblicherweise ein Validierungsdatenset. Das ist im Prinzip eine Menge von Messungen / Feature-Vektoren mit bekannten Klassen, die jedoch nicht zum Trainieren verwendet wurden. Dann lässt man die trainierte SVM die Validierungsdaten klassifizieren und vergleicht das Ergebnis mit den bekannten Klassen. Durch den Klassifizierungsfehler den die SVM am Validierungsdatenset produziert kann man so abschätzen, wie groß der Fehler für neue Beobachtungen mit unbekannter Klasse sein wird. Die Voraussetzung dafür ist natürlich, dass das Validierungsdatenset

die Grundgesamtheit der möglichen Messungen repräsentiert und insbesondere von den Trainingsdaten unabhängig ist. Sind weitere Details über die Vorkommenshäufigkeiten der Klassen in der Grundgesamtheit bekannt, kann man das Validierungsdatenset auch dahingehend stratifiziert werden.

Oft hat man jedoch die Situation nicht ausreichend viele Daten (mit bekannten Klassen) zu haben, um einen Teil für die Validierung zu verwenden. In diesem Fall bietet sich *k-fold Kreuzvalidierung* an. Dabei wird das Trainingsdatenset zufällig in k (etwa) gleich große Segmente aufgeteilt. Dann werden k SVMs trainiert, die jeweils nur die Daten aus $k-1$ Segmenten zum Trainieren benutzen und das verbleibende Segment als Validierungsdatenset verwenden. Die resultierenden k Fehlerraten können dann zu einem Gesamtergebnis gemittelt werden.

4.2. Klassifizierung der Testdaten

Mit den Feature-Vektoren der Messdaten (genauer: wieder deren Geschwindigkeitskanal) aus Abschnitt 2 wurde nun eine Support-Vektor-Maschine mit Gauß-Kernel im C-Classification Modus trainiert. Die Bewertung dieser erfolgte mittels 2-fold Kreuzvalidierung. Damit konnten nun die verschiedenen Varianten der SAX-Methode miteinander verglichen werden. Betrachtet wurden als erstes von der bag-of-patterns Methode erzeugte Feature-Vektoren mit den SAX-Parametern Fensterlänge 96, Wortlänge $w = 6$ und Alphabetgröße $a = 4$. Allerdings wurden 3 SAX-Varianten zur Erzeugung der Feature-Vektoren benutzt:

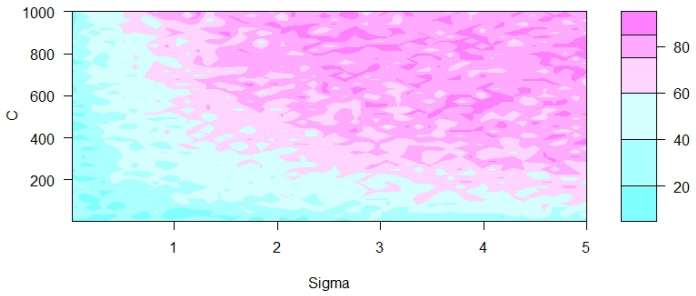
1. bezüglich Mittelwert und Standardabweichung normalisierte Fenster
2. nicht normalisierte Fenster (Breakpoints wie in Kapitel 2)
3. nicht normalisierte Fenster und tf-idf Gewichtung

Diese Varianten sollten untereinander verglichen werden. Dazu galt es, für jede Variante die passenden Parameter zu finden, nämlich σ für den Gauß-Kernel, und C als Schranke für den Einfluss einzelner Beobachtungen. Dazu wurde σ im Intervall $[0.01, 5]$ und C im Intervall $[5, 1000]$ variiert und für jede Kombination die Performance der SVM bestimmt. Die Ergebnisse für die Gruppen sind in Abbildung 4.1 dargestellt.

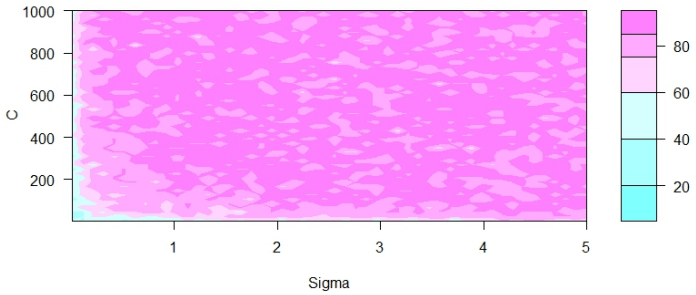
In diesen Abbildungen ist bereits zu sehen, dass alle 3 Varianten mit den richtigen Einstellungen gute Klassifikationsraten um die 85% erreichen. Dabei scheint die tf-idf Variante am robustesten gegenüber einer falschen Parameterwahl zu sein, während bei der klassischen, normalisierten Variante vor allem zu kleine σ Werte zu schlechten Ergebnissen führen.

Es stellte sich nun die Frage, wie viele Komponenten der Feature-Vektoren notwendig sind um noch ein gutes Klassifikationsergebnis zu erhalten. Um das herauszufinden wurden zunächst für jede der drei Gruppen der Feature-Vektoren passende Einstellungen aus den Plots abgelesen und fixiert. Das waren:

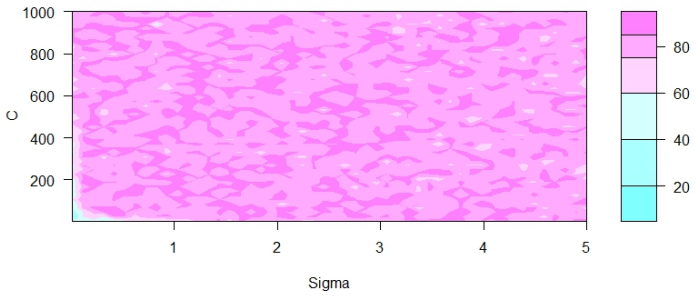
- normalisiert: $C = 600$, $\sigma = 4$
- nicht normalisiert: $C = 200$, $\sigma = 2$
- nicht normalisiert; tf-idf: $C = 200$, $\sigma = 2$



Normalisierte SAX-Fenster.



Nicht normalisierte SAX-Fenster.



Nicht normalisierte SAX-Fenster und tf-idf Gewichtung.

Abbildung 4.1.: Anteil der richtig klassifizierten Messungen gegen die SVM-Parameter.

Dann wurden die Komponenten der Feature-Vektoren (für jede Gruppe extra) nach ihrer Diskriminanzfähigkeit sortiert; das heißt es wurde das Verhältnis der totalen Varianz in einer Komponente über alle Klassen zur mittleren Varianz innerhalb der Klassen betrachtet. Ist die totale Varianz groß, aber die Varianz innerhalb der Klassen klein, ist die jeweilige Komponente gut zur Diskriminierung geeignet. Dann wurden SVMs trainiert, deren Trainingsdaten jedoch nur die jeweils d besten Komponenten enthielten, wobei d Werte im Bereich von 1 bis 4050 annahm. Für jeden Wert von d wurden 10 SVMs trainiert. Die entstehende Punktwolke wurde mit der LOWESS-Methode [Cleveland, 1979] geglättet (Abbildung 4.2).

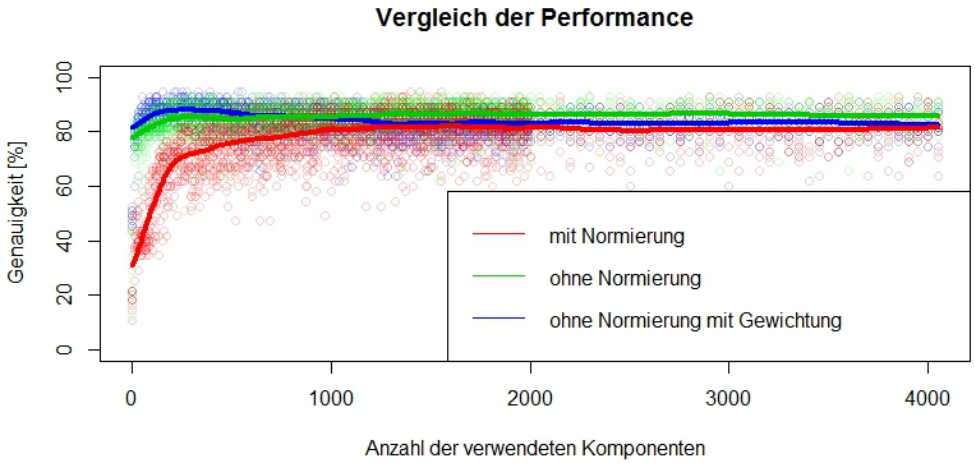


Abbildung 4.2.: Anteil der richtig klassifizierten Punkte gegen die Anzahl der verwendeten Komponenten.

Man sieht im Plot, dass bereits etwa 250 Komponenten ausreichen (Maximum der blauen LOWESS-Linie bei 241) um gute Klassifizierungsergebnisse zu erhalten. Tatsächlich wird vor allem bei den gewichteten Feature-Vektoren die Performance sogar wieder schlechter wenn mehrere Komponenten verwendet werden. Das ist aber auch nicht besonders verwunderlich, denn tf-idf gewichtet die Komponenten ja nach ihrer „Aussagekraft“, also sind „schlechte“ Komponenten alle sehr nahe bei 0 und damit ist ihre Gesamtvarianz klein und sie werden bei der obigen Prozedur erst spät hinzugefügt.

Nun wurden —aus den selben Daten— auch noch ganz neue Feature-Vektoren mit anderen SAX-Parametern erzeugt. Die verwendeten Parameter waren eine Fensterlänge von 40, eine Wortlänge von $w = 4$ und $a = 4$ Symbole. Mit diesen Einstellungen erhält man viel kürzere Feature-Vektoren, nämlich solche mit einer Dimension von $4^4 = 256$. Es wurden analog zum vorigen Experiment wieder 3 Gruppen erzeugt, die sich bezüglich Normalisierung der Fenster und Gewichtung der Komponenten unterschieden. Auch hier wurden wieder die passenden SVM-Parameter für jede Gruppe gesucht. In den Abbildungen zeigt sich dahingehend ein ganz ähnliches Bild wie bei den vorigen SAX-Parametern, weshalb die vorher verwendeten SVM-Parameter auch beibehalten wurden. Mit diesen wurden dann wie vorhin für wachsende Anzahlen an Feature-Vektor Komponenten die Performance verglichen (Abbildung 4.3).

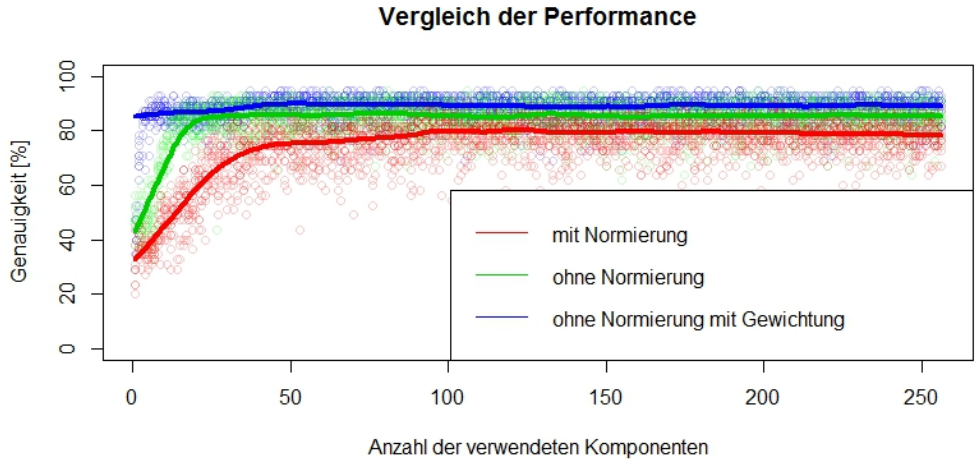


Abbildung 4.3.: Anteil der richtig klassifizierten Punkte gegen die Anzahl der verwendeten Komponenten; kurze Feature-Vektoren.

Überraschenderweise zeigte sich eine praktisch gleich gute, wenn nicht sogar etwas bessere Performance wie bei den langen Feature-Vektoren. Unter Verwendung von tf-idf scheinen sogar schon etwa 50 Komponenten auszureichen um die Messungen gut zu klassifizieren. Im Scatterplot in Abbildung 4.4 sind die besten 3 Komponenten der tf-idf Gruppe zu sehen. Es fällt auf, dass die drei korrespondierenden SAX-Wörter relativ konstantes Verhalten beschreiben. Auch optisch sind die einzelnen Gruppen dabei schön getrennt.

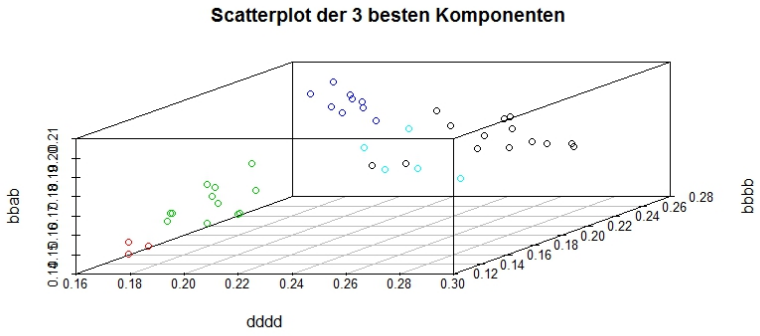


Abbildung 4.4.: Die besten 3 Komponenten der Feature-Vektoren. SAX-Fenster nicht normalisiert, Komponenten mit tf-idf gewichtet.

5. Verwendung von mehreren Messkanälen

Bis jetzt wurde für die Klassifizierung der Messungen immer die Fahrzeuggeschwindigkeit betrachtet und andere Messkanäle ignoriert. Das soll nun geändert werden; insbesondere die Messkanäle der Motordrehzahl und des Motordrehmoments sollen in den Feature-Vektor integriert werden, da diese —zusammen mit der bereits verwendeten Fahrzeuggeschwindigkeit— gemeinhin als die wichtigsten Messkanäle zur Beschreibung einer Fahrt gelten. Verwendet wurden die gleichen Daten wie in den vorangegangenen Kapiteln. Drehzahl und Drehmoment wurden dabei ebenfalls mit einer Abtastrate von 1 Hz gemessen. Ein Plot eines typischen Verlaufs ist in Abbildung 5.1 zu sehen.

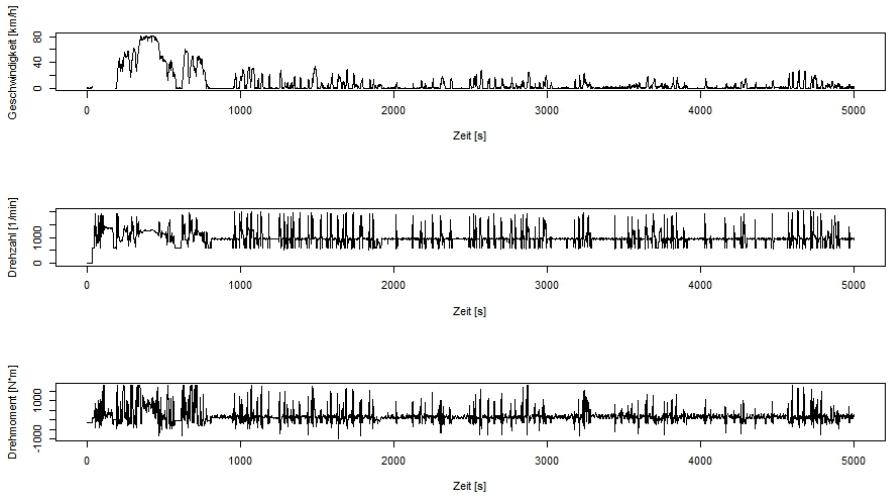


Abbildung 5.1.: Geschwindigkeit, Drehzahl und Drehmoment für eine Messung aus der Gruppe Müllfahrzeuge (Ausschnitt)

5.1. Zusammenhängen der Feature-Vektoren

Nun wurde als erste und einfachste Möglichkeit diese Kanäle in die Klassifikation zu integrieren für jeden Kanal getrennt ein Feature-Vektor berechnet und diese drei kleinen Vektoren zu einem großen Vektor aneinandergereiht. Waren also die 3 kleinen Feature-Vektoren $F_1 = (f_1^{(1)}, \ldots, f_n^{(1)})$, $F_2 = (f_1^{(2)}, \ldots, f_n^{(2)})$ und $F_3 = (f_1^{(3)}, \ldots, f_n^{(3)})$, so ergab sich der Gesamt-Feature-Vektor als $F_G = (f_1^{(1)}, \ldots, f_n^{(1)}, f_1^{(2)}, \ldots, f_n^{(2)}, f_1^{(3)}, \ldots, f_n^{(3)})$. Aufgrund der Beobachtungen im vorigen Kapitel wurden die SAX-Parameter so gewählt, dass kürzere Feature-Vektoren entstehen, also Fensterlänge 40, Wortlänge $w = 4$ und $a = 4$ Buchstaben was eine Länge von 256 bei den Vektoren für die einzelnen Kanäle ergibt, also eine Gesamtlänge von 768. Dies ist noch immer viel kleiner als 4096, was die Länge der

Vektoren für eine Dimension mit den anderen SAX-Parametern war. Die Fenster wurden nicht normalisiert und die zusammengesetzten Vektoren mit tf-idf gewichtet. Die verwendeten Breakpoints waren:

- Geschwindigkeit: 25, 50, 75 [km/h] (unverändert zu vorhin)
- Drehmoment: -1500, 0, 1500 [Nm]
- Drehzahl: 900, 1300, 1800 [1/min]

Dann wurden wieder die Parameter der SVM variiert, wobei sich zeigte, dass die Abhängigkeit von den Parametern durch die Verwendung der 3 Kanäle noch weiter verringert werden konnte. Für fast jede Kombination wurden gute Ergebnisse erzielt (Abbildung 5.2). Aus Gründen der Vergleichbarkeit wurden die Parameter also gleich wie vorhin gewählt, nämlich als $C = 200$ und $\sigma = 2$. Dann wurden wieder die Performance in Abhängigkeit von der Anzahl der verwendeten Komponenten (sortiert nach der Fähigkeit zur Diskriminierung) bestimmt (Abbildung 5.3). Auch hier zeigt sich wieder die beste Performance (93.11%) bei der Verwendung von etwa 50 Komponenten. Unter den besten 50 Komponenten stammten 11 von der Fahrzeuggeschwindigkeit, 24 vom Drehmoment und 15 von der Drehzahl. Unter den besten 3 stammten 0 von der Fahrzeuggeschwindigkeit, 2 vom Drehmoment, und 1 von der Drehzahl. Die besten 3 Komponenten entsprachen den Wörtern *bbba* (Drehmoment), *bbdd* (Drehmoment) und *cddd* (Drehzahl).

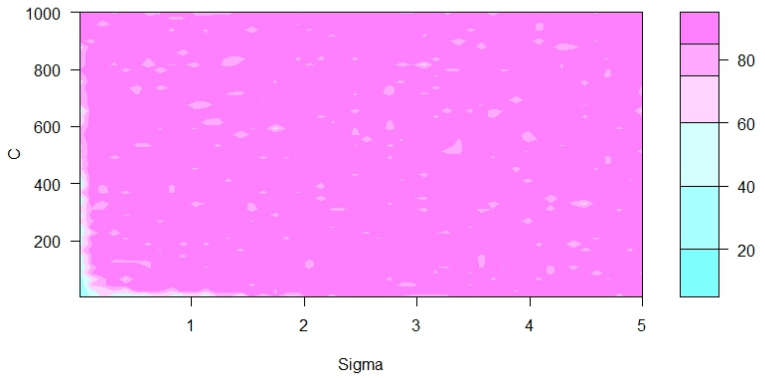


Abbildung 5.2.: Anteil der richtig klassifizierten Punkte unter Verwendung von 3 Messkanälen - Feature-Vektoren zusammengehängt

Durch Verwendung dieser Methode kann man also die Performance verbessern, ohne die Dimension stark zu vergrößern, die beste Performance ergibt sich ja noch immer bei der Verwendung von etwa 50 Komponenten.

5.2. Verwendung der ersten Hauptkomponente

Als zweiter Ansatz wurde eine Idee versucht die von Tanaka et al. [2005] vorgestellt wurde. Dort wird eine Messung durch die erste Hauptkomponente ihrer zu verwendenden Messkanäle repräsentiert; wir betrachten also eine Linearkombination der 3 Messkanäle. Bei

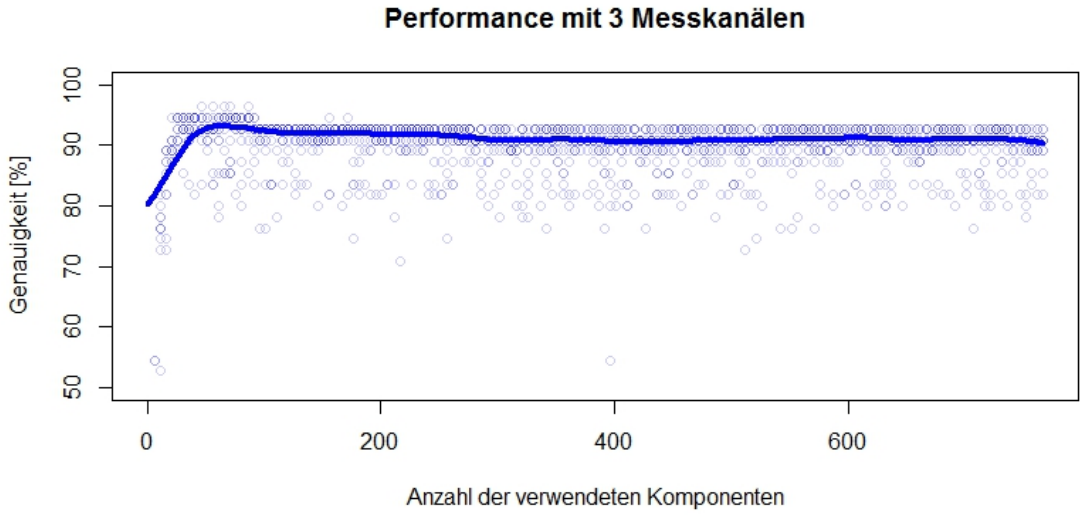


Abbildung 5.3.: Anteil der richtig klassifizierten Messungen gegen die Anzahl der verwendeten Komponenten; Feature-Vektoren zusammengehängt

Tanaka et al. [2005] geht es jedoch nur um das Erkennen von Mustern in einer einzelnen Zeitreihe. Nachdem sich die Koeffizienten der Linearkombination mit jeder Messung ändern ist es fraglich ob diese Methode auch zur Klassifizierung geeignet ist. Wir wollen es jedoch versuchen. Mit den gleichen SAX-Parametern wie vorhin erzeugen wir jetzt also Feature-Vektoren der jeweils ersten Hauptkomponente der 3 Kanäle. Die Fenster werden normalisiert, die tf-idf Gewichtung wird jedoch verwendet. Dann wurden wieder die SVM Parameter variiert (Abbildung 5.4). Dabei zeigte sich jedoch schon, dass die Rate der korrekt klassifizierten Messungen weit unter jener der anderen Methoden liegt, weshalb diese Methode nicht weiter verfolgt wurde.

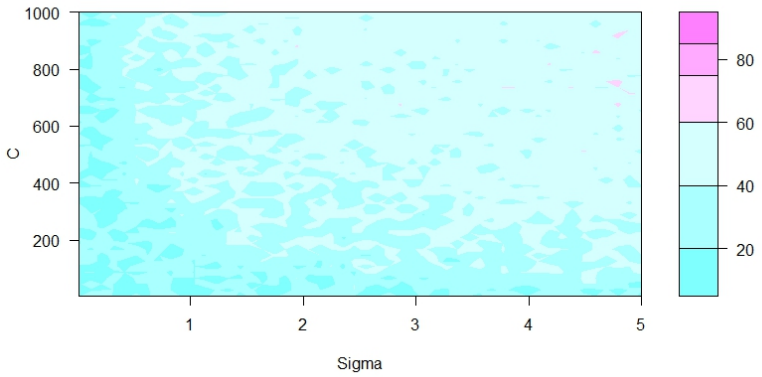


Abbildung 5.4.: Performance für verschiedene SVM-Parameter bei Verwendung der ersten Hauptkomponente der 3 Kanäle

5.3. Höherdimensionale SAX-Repräsentation

Bei der Aneinanderreihung der Feature-Vektoren im Abschnitt 5.1 geht durch die vollkommen getrennte Betrachtung der Messkanäle natürlich jede Information über Gleichzeitigkeit verloren. Um das zu ändern wurde die SAX-Methode adaptiert. Im originalen SAX-Verfahren wird ja die y-Achse in mehrere (in unserem Fall: in 4) Segmente unterteilt, denen jeweils ein Symbol zugewiesen wird. Wir wollen nun die 3 y-Achsen der Messkanäle simultan in ihre Segmente unterteilen und jeder auftretenden Kombination ein eigenes Symbol geben. Die Idee wird für zwei Dimensionen in Abbildung 5.5 verdeutlicht.

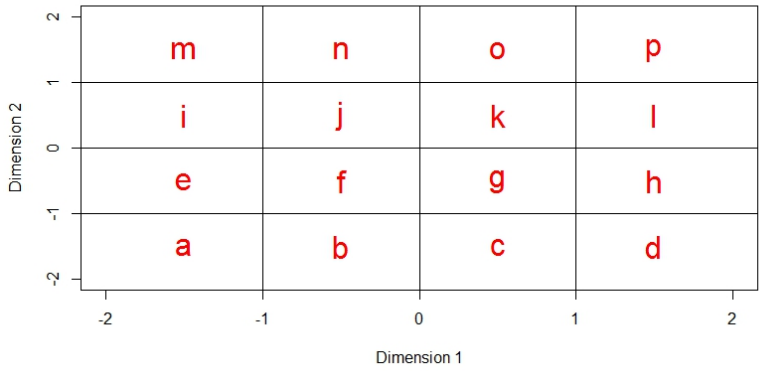


Abbildung 5.5.: Vorgehensweise bei der Ermittlung des SAX-Symboles in 2 Dimensionen.

Dieses Vorgehen ist natürlich komplett äquivalent dazu, dass wir nicht die Anzahl der Symbole von 4 auf 4^3 erhöhen, sondern die Wortlänge von 4 auf 12 verdreifachen. Jedes Drittel des langen Wortes entspricht dann dem ursprünglichen SAX-Wort für die jeweilige Dimension. In beiden Fällen gibt es dann $(4^3)^4 = 4^{(4\cdot 3)} = 16777216$ mögliche Wörter. Die entsprechenden Feature-Vektoren haben damit natürlich eine viel zu große Dimension um damit sinnvoll eine SVM zu trainieren. Tatsächlich war es nicht einmal möglich alle Feature-Vektoren auf einmal im Hauptspeicher zu halten. Es zeigte sich aber, dass nur sehr wenige Kombinationen überhaupt vorkamen und damit sehr viele Komponenten der Feature-Vektoren 0 waren. Deshalb wurden immer nur einige Vektoren in den Speicher geladen um zu bestimmen welche Indizes in allen Vektoren gleich 0 waren. War dies bekannt, wurden diese Komponenten sofort gelöscht, womit sich die Dimension der Vektoren auf 31005 reduzierte und sie in den Speicher passten, womit sinnvoll mit ihnen gearbeitet werden konnte. Auch mit dieser Länge war es aber noch nicht möglich in sinnvoller Zeit (und mit im Verhältnis zur Dimension geringer Datenanzahl) eine SVM zu trainieren um die besten Parameter zu finden. Darum wurden die Parameter $C = 200$ und $\sigma = 2$ von vorher weiterverwendet. Dann wurden wieder die Komponenten sortiert und die Performance mit 1 bis 2000 Komponenten bewertet (Abbildung 5.6).

In Abbildung 5.7 ist ein Vergleich der verschiedenen Methoden mit bis zu 250 Komponenten zu sehen. In allen 3 Fällen wurde die nicht normalisierte tf-idf Variante gewählt, mit Breakpoints wie oben. Es ist ersichtlich, dass die beiden Varianten die 3 Messkanäle verwenden in etwa gleich gut abschneiden, während die Variante die nur auf Geschwindigkeit basiert etwas schlechter ist. Außerdem scheinen in diesem Datenset einige Beobachtungen

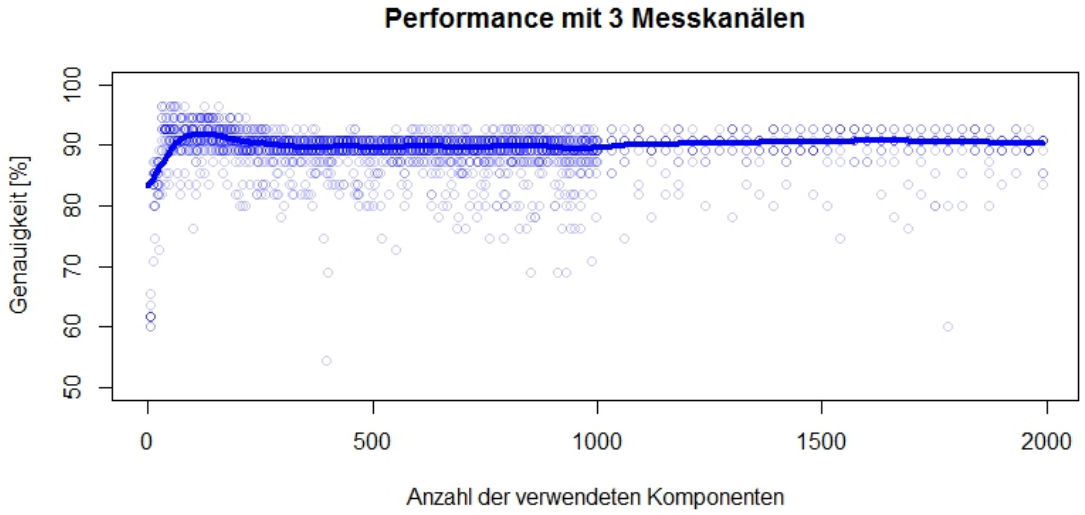


Abbildung 5.6.: Rate der richtig klassifizierten Messungen gegen die Anzahl der verwendeten Komponenten. Erzeugung der Feature-Vektoren mit mit SAX in 3 Dimensionen.

sehr untypisch für ihre Klasse zu sein, nachdem noch mit keiner der Methoden 100% der Beobachtungen richtig klassifiziert wurden.

Zuletzt wurde hier noch versucht eine Hauptkomponentenanalyse auf die besten 50 Komponenten der Feature-Vektoren anzuwenden. Dies führte ebenfalls zu einem zufriedenstellenden Ergebnis wie der Plot auf die ersten beiden Hauptkomponenten, in dem die korrekten Gruppen schön getrennt sind, zeigt (Abbildung 5.8).

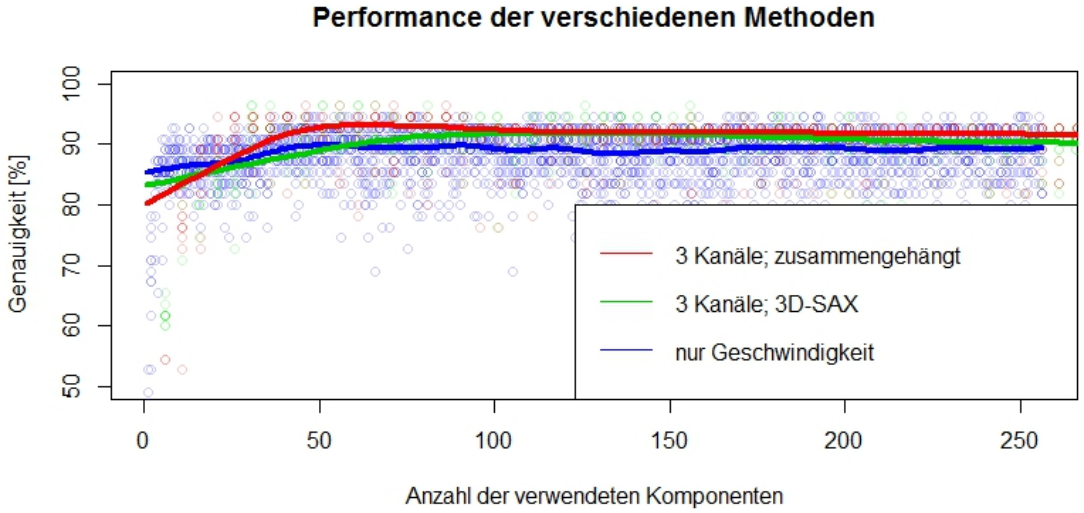


Abbildung 5.7.: Vergleich der Performance von 3 Methoden.

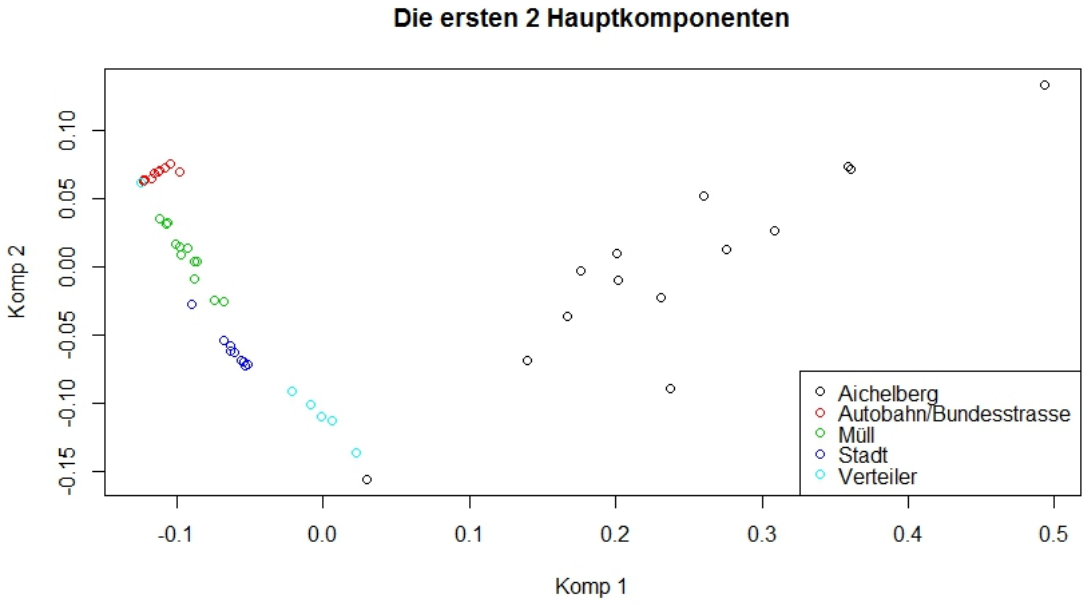


Abbildung 5.8.: Plot der Datenpunkte auf die ersten beiden Hauptkomponenten.

6. Drei Methoden zum Finden von Clustern

Die gute Performance der trainierten SVMs legt nahe, dass die Messungen durch Verwendung der SAX-Methode gut repräsentiert werden können. Im nächsten Schritt möchte man nun die Messungen oder Teile davon clustern um Gruppen in diesen zu finden. Dazu sollen nun drei ausgewählte Clustering-Methoden näher vorgestellt werden. Die Feature-Vektoren werden dazu als Datenpunkte in einem hochdimensionalen Raum betrachtet. Jeder Feature-Vektor ist ein Punkt, dessen Koordinaten den verschiedenen SAX-Wörtern entsprechen. Als Basis für alle vorgestellten Clustermethoden sei eine Metrik $d(\cdot, \cdot)$ auf dem Raum der Datenpunkte fixiert.

6.1. Hierarchisches Clustern

Beim hierarchischen Clustern geht man von einer Hierarchiestruktur in den Gruppen aus. Die gesamte Menge der Daten bildet die oberste Ebene, also den größten Cluster. Diesen teilt man nun in 2 Subcluster, die ihrerseits wiederum geteilt werden. Dies setzt sich solange fort, bis alle Cluster aus einzelnen Datenpunkten bestehen und somit nicht mehr weiter geteilt werden können. Abbildung 6.1 verdeutlicht das Prinzip einer Hierarchiestruktur.

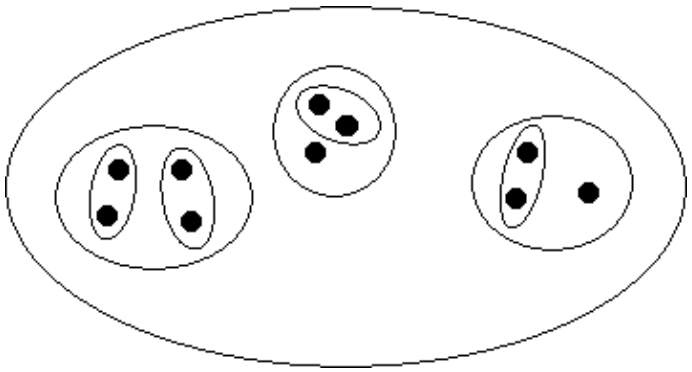


Abbildung 6.1.: Visualisierung der hierarchischen Betrachtungsweise. Jeder Cluster besteht wieder aus Subclustern (die auch aus nur einem Datenpunkt bestehen können).

Anstatt nun die Gesamtmenge der Daten immer weiter zu partitionieren, kann man auch von der umgekehrten Richtung aus starten, jeden Datenpunkt als einzelnen Cluster betrachten und sukzessive die beiden zueinander nächsten Cluster zusammenfügen. Man spricht dann von agglomerativen hierarchischen Clustern. Dazu muss natürlich ein Abstandsmaß für Cluster definiert werden. Übliche einfache Abstandsdefinitionen zwischen zwei Clustern C_1 und C_2 sind:

- $\min_{a \in C_1, b \in C_2} d(a, b)$ (single-linkage)

- $\max_{a \in C_1, b \in C_2} d(a, b)$ (complete-linkage)
- $\frac{1}{|C_1||C_2|} \sum_{a \in C_1, b \in C_2} d(a, b)$ (average-linkage)

Es existieren aber auch komplizierte Varianten wie etwa *Ward's distance* [Ward Jr, 1963], die (bei Verwendung der euklidischen Metrik) der Zunahme der Varianz beim Vereinigen der beiden Cluster entspricht, und sich als

$$\frac{d(\bar{a}, \bar{b})}{\frac{1}{|C_1|} + \frac{1}{|C_2|}}$$

ergibt. Dabei sind $\bar{a}$ und $\bar{b}$ die Zentren der Cluster C_1 und C_2, im euklidischen Fall also die Mittelwerte der Punkte im Cluster.

Hat man die Hierarchie der Cluster bestimmt, so wird das Ergebnis meist in einem Dendrogramm dargestellt. Das Dendrogramm ist ein Binärbaum, dessen Wurzel die Gesamtmenge der Daten darstellt. Jeder Kindknoten ist ein Cluster, der wiederum alle seine Kindknoten als Subcluster enthält. Die Blätter entsprechen schlussendlich den einzelnen Datenpunkten. Als zusätzliches Attribut ist im Dendrogramm die Länge der Verbindungslinie eines Knoten zu seinen Kindknoten festgelegt. Diese entspricht gerade der (Cluster-) Distanz zwischen den Kindknoten. Man spricht hier von der *Höhe* des Dendrogramms. Nun kann man den gewünschten Maximalabstand zwischen Clustern festlegen, und das Dendrogramm auf dieser Höhe „abschneiden". Alle nun entstehenden Wurzelknoten auf geringerer Höhe definieren die gewünschten Cluster. Zur Verdeutlichung wurde in Abbildung 6.2 das Dendrogramm der Feature-Vektoren der Messungen aus dem vorhergehenden Abschnitt erstellt.

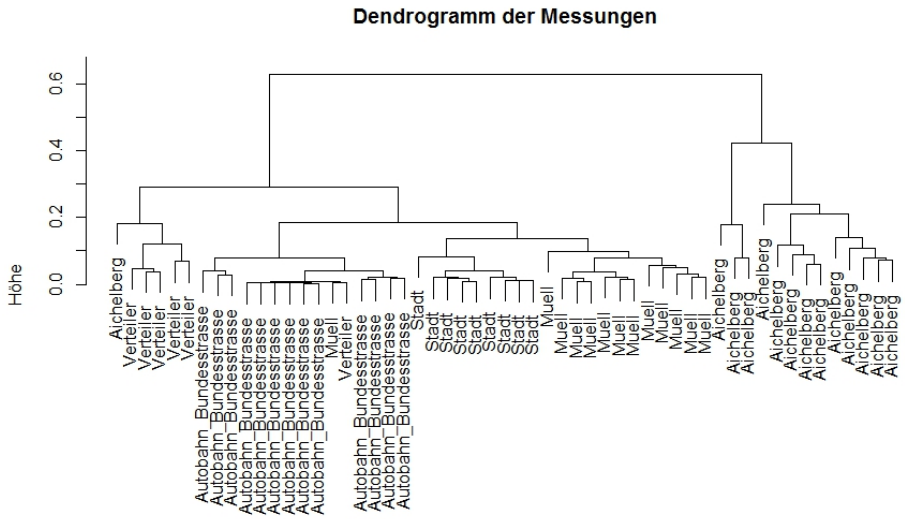


Abbildung 6.2.: Hierarchisches, agglomeratives, complete-linkage Clustern der Messdaten mit eukidscher Distanz. Die Bezeichnungen entsprechen den wahren Klassen.

6.2. Partitioning Around Medoids

Der Partitioning Around Medoids (PAM) Algorithmus wurde erstmals von Rousseeuw und Kaufman [1987] vorgestellt. Im Gegensatz zum hierarchischen Clustermodell wird hier von einem streng partitionierenden Modell ausgegangen, das heißt, jeder Datenpunkt gehört genau zu einem Cluster und dieser Cluster hat weder Subcluster noch ist er selbst Teil von größeren Clustern. Ein potentieller Nachteil von PAM ist, dass die Anzahl k der vorhandenen Cluster im Voraus bekannt sein muss. Ein Vorteil ist jedoch, dass der Algorithmus für jeden Cluster ein *Medoid* liefert. Das Medoid ist (je nach verwendeter Metrik) mit dem Mean oder Median verwandt, charakterisiert also die Lage der Daten eines Clusters. Im Gegensatz zu anderen Lagemaßen ist es jedoch selbst immer Teil der Daten. Es wird als jener Datenpunkt definiert, dessen mittlere Distanz zu allen anderen Datenpunkten im Cluster minimal ist. Der PAM-Algorithmus selbst ist eng verwandt mit dem *k-means* Algorithmus und läuft so ab:

1. Wähle k der Datenpunkte als Medoide. Das kann zufällig oder mit Hilfe einer Heuristik erfolgen.
2. Weise die restlichen Datenpunkte jenem Medoid zu das ihnen am nächsten liegt.
3. Für jedes Medoid:
   - Für jeden Datenpunkt der kein Medoid ist:
     - Mache den Datenpunkt zum neuen Medoid (und das aktuelle Medoid zum „normalen“ Datenpunkt) und berechne die Kosten dieser Konfiguration, also die Summe der Abstände aller Punkte zu ihrem jeweiligen Medoid.
4. Führe jenen Austausch in (3.) durch, der die geringsten Kosten produziert.
5. Führe die Schritte (2.) -(4.) solange durch, bis sich die Auswahl der Medoide nicht mehr ändert.

Zur Veranschaulichung wurden wieder die Feature-Vektoren aus dem vorigen Abschnitt verwendet und mit dem PAM-Algorithmus geclustert. Die Implementation des Algorithmus stammt aus dem `R`-Package `cluster` [Maechler et al., 2014]. In Abbildung 6.3 ist ein Plot des Ergebnisses auf die ersten beiden Hauptkomponenten zu sehen.

Bestimmung der Anzahl der Cluster: Wie schon erwähnt, muss für die Ausführung des PAM-Algorithmus die Anzahl der vorhandenen Cluster im Voraus bekannt sein. Da dies in der Praxis aber sehr selten der Fall ist, führt man den Algorithmus meist mit verschiedenen Werten für k aus und vergleicht dann die Qualitäten der entstandenen Clusterings. Eine Kenngröße um diesen Vergleich durchzuführen ist der *Silhouettenkoeffizient* [Rousseeuw, 1987], der wie folgt berechnet werden kann:

Gegeben seien k Cluster $C_1, \ldots, C_k$ die insgesamt n Datenpunkte enthalten. Der i-te Cluster enthalte n_i Punkte und sei gegeben als $C_i = \{y_1^{(i)}, \ldots, y_{n_i}^{(i)}\}$ und es sei $\sum_{i=1}^{k} n_i = n$. Die Silhouette eines Punktes $y_j^{(i)}$ im i-ten Cluster ist definiert als

$$s(y_j^{(i)}) = \frac{b(y_j^{(i)}) - a(y_j^{(i)})}{\max\{a(y_j^{(i)}), b(y_j^{(i)})\}}.$$

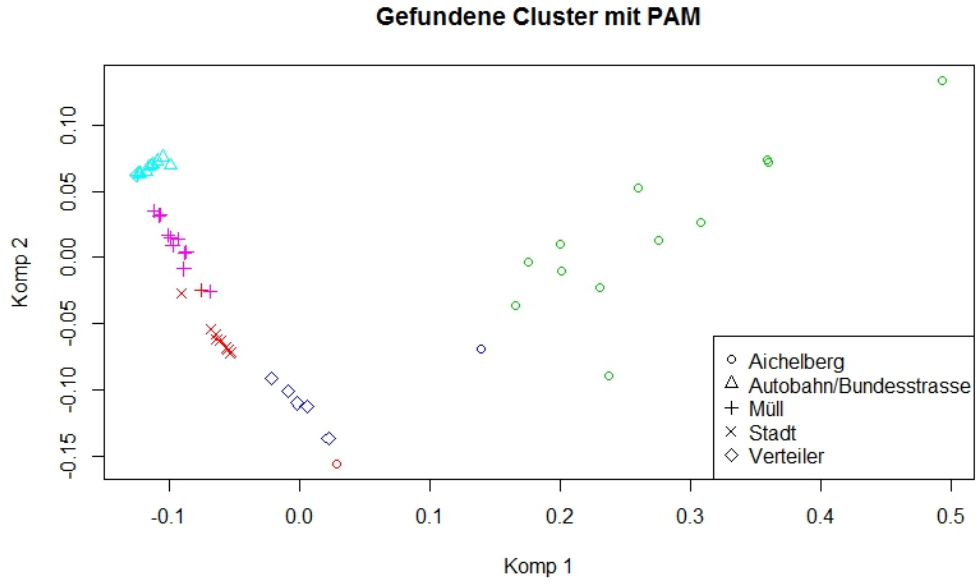


Abbildung 6.3.: Ergebnis des PAM-Algorithmus auf die ersten beiden Hauptkomponenten der Feature-Vektoren geplottet. Die Farbe gibt die von PAM gefundene Zugehörigkeit an, das Symbol die wahre Klasse.

Dabei misst $a(\cdot)$, wie gut $y_j^{(i)}$ in seinem eigenen Cluster liegt und ist als mittlerer Abstand zu allen anderen $n_i - 1$ Punkten im Cluster definiert, also als

$$a(y_j^{(i)}) = \frac{1}{n_i - 1} \sum_{\substack{m=1 \\ m \neq j}}^{n_i} d(y_m^{(i)}, y_j^{(i)}).$$

Der Wert von $b(\cdot)$ misst wie gut $y_j^{(i)}$ von Punkten in anderen Clustern getrennt ist und ist als das Mittel der Abstände zu Punkten aus anderen Clustern definiert:

$$b(y_j^{(i)}) = \frac{1}{n - n_i} \sum_{\substack{l=1 \\ l \neq i}}^{k} \sum_{m=1}^{n_l} d(y_m^{(l)}, y_j^{(i)}).$$

Damit ist klar, dass $-1 \leq s(\cdot) \leq 1$ hält und große Werte von $s(\cdot)$ eine gute Zuordnung des Datenpunktes nahelegen.

Der Silhouettenkoeffizient eines Clusterings ergibt sich nun als das arithmetische Mittel der Silhouetten aller Datenpunkte, also als

$$sk(y_j^{(i)}) = \frac{1}{n} \sum_{i=1}^{k} \sum_{j=1}^{n_i} s(y_j^{(i)}).$$

Es ist sofort aus der Definition klar, dass der Silhouettenkoeffizient nur dann definiert ist, wenn die Anzahl der Cluster $k \geq 2$ ist. Insbesondere kann mit dem Silhouettenkoeffizienten nicht festgestellt werden ob die Daten aus nur einem Cluster bestehen.

Beispiel. Als Beispiel wenden wir den PAM-Algorithmus mit $k = 1, \ldots, 7$ auf die ersten 2 Hauptkomponenten der Feature-Vektoren unserer 5 Gruppen (vgl. Abbildung 6.3) an und berechnen die zugehörigen Silhouettenkoeffizienten. Es ergibt sich:

k	2	3	4	5	6	7
Silhouettenkoeffizient	0.60	0.60	0.64	0.65	0.56	0.54

Der größte Silhouttenkoeffizient taucht hier bei $k = 5$ Clustern auf, was der wahren Gruppenanzahl entspricht.

$\Diamond$

6.3. Modellbasiertes Clustern mit dem GMM

Als dritte Methode zum Clustern von Daten soll modellbasiertes Clustern mit dem *Gaussian Mixture Model* (GMM) vorgestellt werden. Dabei geht man davon aus, dass die Verteilung der Daten einer Mischung von multivariaten Normalverteilungen entspricht, also deren Dichte durch

$$f(y) = \sum_{i=1}^{k} \pi_i \mathcal{N}_{\mu_i, \Sigma_i}(y), \tag{6.1}$$

mit $\sum_{i=1}^{k} \pi_i = 1$ und $\pi_i \geq 0$, gegeben ist.

Sind die Parameter dieser Dichte bekannt, so kann man mit dem Satz von Bayes für jede der k Komponenten die a posteriori Wahrscheinlichkeit berechnen, dass eine Beobachtung y aus eben dieser Komponente stammt. Um nun die Daten zu clustern, wird jede Beobachtung jener Komponente zugeordnet, von der sie mit der größten Wahrscheinlichkeit erzeugt wurde. Die Anzahl der Cluster entspricht damit also k und muss auch hier im Voraus bekannt sein.

Will man jedoch den Maximum-Likelihood-Schätzer (ML-Schätzer) der Parameter für dieses Modell berechnen, so schlägt dies fehl, denn die beobachteten Daten enthalten nicht alle notwendigen Informationen. Konkret müssten wir bereits im Voraus wissen, von welcher Komponente ein Datenpunkt erzeugt wurde, um die ML-Schätzer berechnen zu können. Eine Lösung für dieses Problem bietet der Expectation-Maximization (EM) Algorithmus der von Dempster et al. [1977] vorgestellt wurde und dessen grundsätzliche Funktionsweise hier noch einmal dargestellt werden soll.

6.3.1. Der EM-Algorithmus

Der EM Algorithmus liefert ganz allgemein eine Approximation an die ML-Schätzer für Modelle in denen die Daten eine nicht beobachtbare Komponente enthalten. In unserem Fall wäre das also die Information über die Zugehörigkeit zu einer bestimmten Komponente. Der Algorithmus läuft dann zweistufig ab: Im E-Schritt berechnet man den, auf den beobachtbaren Teil der Daten bedingten, Erwartungswert der gemeinsamen Dichte für festgehaltene Modellparameter. Im darauffolgenden M-Schritt berechnet man die ML-Schätzer der Parameter anhand des vorher bestimmten Erwartungswertes und der beobachteten Daten. Diese Prozedur wird bis zur Konvergenz wiederholt, wobei die bei

der Maximierung gefundenen Parameter immer für den jeweils darauffolgenden E-Schritt verwendet werden.

Formal haben wir folgendes: Seien Daten gegeben, die aus einem beobachtbaren Teil y und einem nicht beobachtbaren Teil z bestehen und die einer Verteilung mit Dichte $f(y, z|\theta)$ folgen. Unser Ziel ist es die (marginale) log-likelihood der beobachteten Teile über den Parametern θ zu maximieren, wir suchen also $\arg\max_\theta \log f(y|\theta)$.

Wir betrachten dafür die auf y bedingte Dichte von z, für die gilt:

$$f(z|y,\theta) = \frac{f(y,z|\theta)}{f(y|\theta)} \Leftrightarrow \log f(y,z|\theta) = \log f(z|y,\theta) + \log f(y|\theta).$$

Wir fixieren einen Parametervektor θ_0, wenden den auf y und θ_0 bedingten Erwartungswert auf die obige Gleichung an, und führen zwei neue Funktionen ein:

$$\underbrace{\mathbb{E}[\log f(y,z|\theta) \mid y,\theta_0]}_{=:Q_{\theta_0}(\theta)} = \underbrace{\mathbb{E}[\log f(z|y,\theta) \mid y,\theta_0]}_{=:H_{\theta_0}(\theta)} + \underbrace{\mathbb{E}[\log f(y|\theta) \mid y,\theta_0]}_{=\log f(y|\theta),\text{ da konstant}}. \tag{6.2}$$

Die Berechnung der Funktion $Q_{\theta_0}(\theta)$ für gegebenes θ_0 ist der E-Schritt. Die anschließende Maximierung von Q_{θ_0} über θ ist der M-Schritt. Wir zeigen nun, dass die Maximierung von Q tatsächlich auch zur Maximierung von $\log f(y|\theta)$ führt, die wir ja eigentlich erreichen wollen. Wir wollen also zeigen, dass für einen im M-Schritt gefundenen Parametervektor θ^* gilt:

$$\log f(y|\theta^*) - \log f(y|\theta_0) \overset{(6.2)}{=} Q_{\theta_0}(\theta^*) - Q_{\theta_0}(\theta_0) - [H_{\theta_0}(\theta^*) - H_{\theta_0}(\theta_0)] \overset{z.Z.}{\geq} 0.$$

Nachdem wir im M-Schritt die Funktion Q_{θ_0} maximiert haben, gilt jedenfalls $Q_{\theta_0}(\theta^*) - Q_{\theta_0}(\theta_0) \geq 0$, es reicht also zu zeigen, dass $H_{\theta_0}(\theta^*) - H_{\theta_0}(\theta_0) \leq 0$ hält. Dies folgt aber direkt aus der Jensen Ungleichung für Erwartungswerte:

$$\begin{aligned}
H_{\theta_0}(\theta^*) - H_{\theta_0}(\theta_0) &= \mathbb{E}[\log f(z|y,\theta^*) - \log f(z|y,\theta_0) \mid y,\theta_0] = \\
&= \mathbb{E}\left[\log \frac{f(z|y,\theta^*)}{f(z|y,\theta_0)} \;\middle|\; y,\theta_0\right] \leq \\
&\leq \log \mathbb{E}\left[\frac{f(z|y,\theta^*)}{f(z|y,\theta_0)} \;\middle|\; y,\theta_0\right] = \\
&= \log \int \frac{f(z|y,\theta^*)}{\cancel{f(z|y,\theta_0)}} \cancel{f(z|y,\theta_0)} \mathrm{d}z = \\
&= \log 1 = 0.
\end{aligned} \tag{6.3}$$

Damit haben wir bereits gezeigt, dass der M-Schritt mit der Maximierung von Q_{θ_0} auch die Funktion $\log f(y|\theta)$ vergrößert (zumindest nicht verkleinert). Als nächstes zeigen wir, dass die Stationarität von Q_{θ_0} auch die Stationarität von $\log f(y|\theta)$ impliziert. Dafür differenzieren wir (6.2) und erhalten:

$$\frac{\partial}{\partial\theta} Q_{\theta_0}(\theta) = \frac{\partial}{\partial\theta} H_{\theta_0}(\theta) + \frac{\partial}{\partial\theta} \log f(y|\theta).$$

Wir erkennen, dass (6.3) nicht nur für den im M-Schritt gefundenen Parametervektor θ^* hält, sondern für beliebige Werte von θ. Damit ist aber θ_0 ein Minimum von H_{θ_0} und es gilt $\frac{\partial}{\partial\theta} H_{\theta_0}(\theta) = 0$. Damit ist das Gewünschte bereits gezeigt.

Als letzter Punkt muss noch gezeigt werden, dass globale Maxima von $\log f(y|\theta)$ auch globale Maxima von Q_{θ_0} sind. Dies folgt aber ebenfalls sofort aus (6.2).

Beispiel. [Die Funktion Q_{θ_0} für das betrachtete Mischmodell] Zum Erkennen von Clustern nehmen wir ja an, dass die Daten einer Mischung von Normalverteilungen folgen (6.1). Wir wollen nun die Funktion $Q_{\theta_0}(\theta)$ für dieses Modell berechnen. Die unbekannten Parameter sind dabei $\theta = (\pi_1, \mu_1, \Sigma_1, \pi_2, \mu_2, \Sigma_2, \ldots, \pi_k, \mu_k, \Sigma_k)$. Der nicht beobachtbare Teil z der Daten enthält genau die Zugehörigkeit der Datenpunkte zu den Komponenten. Wir definieren dazu zu jedem Datenpunkt y_j k-Stück Indikatorvariablen z_{ij}, für die gilt:

$$z_{ij} = \begin{cases} 1 & \text{wenn } y_j \text{ aus Komponente } i \text{ stammt} \\ 0 & \text{sonst} \end{cases} \qquad \forall i = 1, \ldots, k \ \forall j = 1, \ldots, n$$

Damit können wir die gemeinsame Dichte von y und z als

$$f(y, z|\theta) = \prod_{j=1}^{n} \prod_{i=1}^{k} [\pi_i \mathcal{N}_{\mu_i, \Sigma_i}(y_j)]^{z_{ij}}$$

schreiben. Die Funktion $Q_{\theta_0}(\theta)$ ist nun der, auf y und fix gewählte Parameter θ_0 bedingte, Erwartungswert des Logarithmus dieser Dichte, also

$$\begin{aligned} Q_{\theta_0}(\theta) &= \mathbb{E}\left[\log f(y, z|\theta) \mid y, \theta_0\right] = \\ &= \mathbb{E}\left[\sum_{j=1}^{n} \sum_{i=1}^{k} z_{ij} (\log \pi_i + \log \mathcal{N}_{\mu_i, \Sigma_i}(y_j)) \;\middle|\; y, \theta_0\right]. \end{aligned}$$

Durch die Konditionierung sind in diesem Erwartungswert aber nur mehr die z_{ij} zufällig. Für deren Erwartungswert erhält man

$$\mathbb{E}[z_{ij} \mid y, \theta_0] = \mathbb{P}[z_{ij} = 1 \mid y, \theta_0] = \frac{\pi_i \mathcal{N}_{\mu_i, \Sigma_i}(y_j)}{\sum_{l=1}^{k} \pi_l \mathcal{N}_{\mu_l, \Sigma_l}(y_j))} =: w_{ij},$$

womit sich die im M-Schritt zu maximierende Funktion $Q_{\theta_0}(\theta)$ als

$$Q_{\theta_0}(\theta) = \sum_{j=1}^{n} \sum_{i=1}^{k} w_{ij} (\log \pi_i + \log \mathcal{N}_{\mu_i, \Sigma_i}(y_j))$$

ergibt. ◊

6.3.2. Im R-Package `mclust` verfügbare Modelle

Die im vorigen Abschnitt vorgestellte allgemeine Variante des EM-Algorithmus muss vor der Implementierung natürlich noch auf das zu schätzende Modell (etwa das vorhin vorgestellte GMM) spezialisiert werden. In der vorliegenden Arbeit wurde die bereits vorhandene Implementation aus dem R-Package `mclust` [Fraley und Raftery, 2002] genutzt. Dieses

Package kann zusätzlich mit verschiedenen GMMs umgehen, die sich in der Wahl der Kovarianzmatrizen (und damit in der Anzahl der zu schätzenden Parameter) unterscheiden. Die verfügbaren Modelle sollen hier kurz vorgestellt werden, damit die im Package verwendeten Bezeichnungen übernommen werden können.

Die Kovarianzmatrizen der Komponenten werden dafür wie folgt zerlegt: Eine Kovarianzmatrix ist notwendigerweise symmetrisch und hat nur reelle Einträge. Damit ist eine solche Matrix mit orthogonalen Matrizen diagonalisierbar, das heißt, es existiert eine eindeutige Zerlegung der Form $\Sigma = R\tilde{F}R^T$ wobei $RR^T = R^TR = I$ hält und $\tilde{F}$ eine Diagonalmatrix ist, die die Eigenwerte von Σ enthält. Da Kovarianzmatrizen auch positiv definit sind, sind alle Einträge von $\tilde{F}$ positiv und insbesondere nicht 0. Wir können also $\tilde{F} = \operatorname{diag}(\tilde{f}_1, \ldots, \tilde{f}_d)$ mittels $v = \sqrt[d]{\prod_{i=1}^{d} \tilde{f}_i}$ und $F = \frac{1}{v}\tilde{F}$ normieren und erhalten die noch immer eindeutige Zerlegung $\Sigma = vRFR^T$. Für diese gilt:

- Die Matrix R enthält die (normierten) Eigenvektoren von Σ. Diese geben genau die Richtungen der Hauptachsen der Kovarianzellipsoide an.
- Die Einträge der Matrix F sind proportional zu den Längen der Hauptachsen der Kovarianzellipsoide. Die Matrix F bestimmt also die Form dieser.
- Der Skalar v ist proportional zum Volumen der Kovarianzellipsoide.

Mit dieser Zerlegung können wir also Richtung, Form und Volumen der Kovarianzellipsoide der Komponenten im Mischmodell beschreiben. Das `R`-Package `mclust` bietet dabei die folgenden Modelle an, die sich in der Form der Kovarianzmatrizen der Komponenten unterscheiden:

`mclust`-Name	Gestalt der Kovarianzmatrizen $\Sigma_k =$
EII	$vI_{d\times d}$
VII	$v_kI_{d\times d}$
EEI	vF
VEI	v_kF
EVI	vF_k
VVI	v_kF_k
EEE	$vRFR^T$
EEV	$vR_kFR_k^T$
VEV	$v_kR_kFR_k^T$
VVV	$v_kR_kF_kR_k^T$

6.3.3. Die Bewertung des Modells und die Bestimmung von k

Die Modellselektion, und damit die Wahl von k und der Kovarianzstruktur, kann mit Hilfe eines *Informationskriteriums* erfolgen. Das könnte zum Beispiel das von Schwarz [1978] vorgestellte *Bayesian Information Criterion* (BIC) sein. Dieses ist definiert als

$$\mathrm{BIC} = -2\log f(y \mid \theta^*) + n_p \log n,$$

wobei n_p die Anzahl der im Modell vorkommenden Parameter und n die Anzahl der Datenpunkte ist. Der Wert von $f(y \mid \theta^*)$ ist die Likelihood-Funktion des Modells an der

Stelle der gefundenen (optimalen) Parameter. Kleinere Werte von BIC legen also ein besser passendes Modell nahe.

Das R-Package `mclust` kann verschiedene Modelle und den zugehörigen BIC-Wert automatisch berechnen, was an einem Beispiel demonstriert werden soll:

Beispiel. Wir erzeugen eine Stichprobe mit 1000 Datenpunkten der Dimension 2. Jeweils 500 dieser 1000 Datenpunkte werden aus einer bivariaten Normalverteilung simuliert. Wir erwarten also, dass ein Modell mit 2 Komponenten die Daten am besten beschreiben wird. Zunächst sollen die beiden Normalverteilungen die gleiche Kovarianzmatrix besitzen, nämlich

$$\Sigma = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}.$$

Die Mittelwerte seien $\mu_1 = (3, 4)$ und $\mu_2 = (-2, 1)$.

In Abbildung 6.4 ist ein Plot der Stichprobe, sowie die BIC-Werte für verschiedene Modelle zu sehen. Es ist erkennbar, dass für jede Kovarianzstruktur das Modell mit zwei Komponenten den optimalen BIC-Wert liefert. Das „EII“-Modell, das auch der tatsächlichen Kovarianzstruktur entspricht, liefert den global größten BIC-Wert. Die `summary` des `mclust`-Objektes liefert weitere Details:

```
Mclust EII (spherical, equal volume) model with 2 components:

 log.likelihood    n df       BIC       ICL
      -4218.066 1000  6 -8477.578 -8494.808
```

Die 6 Parameter die hier geschätzt wurden sind die Mittelwerte (4 Freiheitsgrade), die 2 Anteile der Komponenten (1 Freiheitsgrad, da die Summe dieser auf 1 normiert ist), sowie der Parameter v der Kovarianzstruktur (1 Freiheitsgrad). Auch die gefundenen Schätzer entsprechen den wahren Parametern sehr gut:

```
> c$parameters$pro
[1] 0.4992177 0.5007823
> c$parameters$mean
         [,1]      [,2]
[1,] 3.018800 -1.902474
[2,] 3.954264 -1.014122
> c$parameters$variance$Sigma
         [,1]     [,2]
[1,] 1.909325 0.000000
[2,] 0.000000 1.909325
```

Als zweiten Test erzeugen wir Daten mit den gleichen Mittelwerten wie vorhin, aber mit einer komplizierteren Kovarianzstruktur, nämlich

$$\Sigma_1 = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}$$

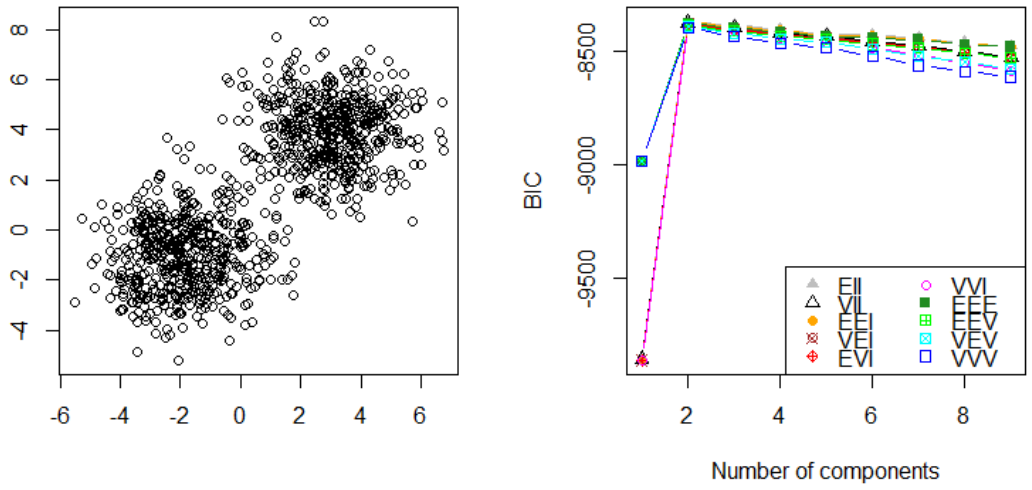


Abbildung 6.4.: Die künstlich erzeugten Daten (2 kreisförmige Cluster gleicher Größe) und die BIC-Werte für verschiedene GMMs.

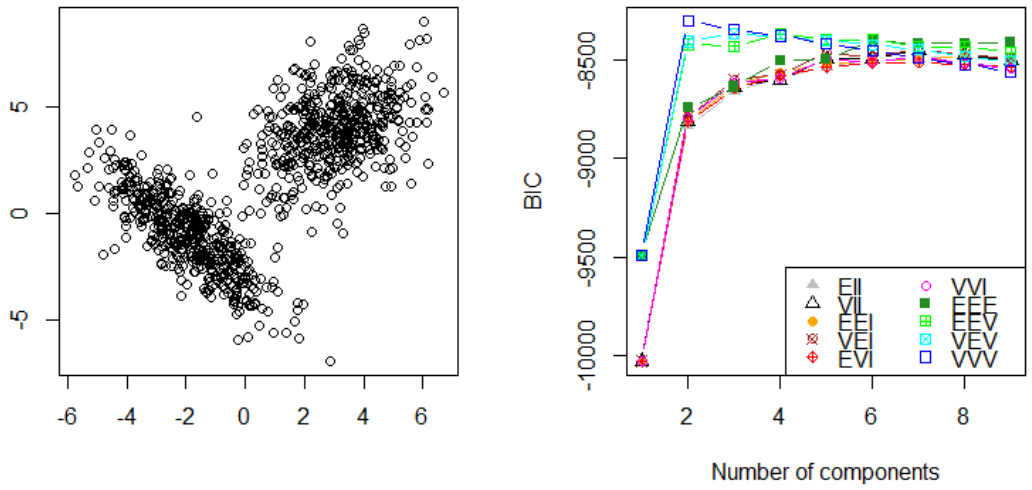


Abbildung 6.5.: Die künstlich erzeugte Daten (2 Cluster unterschiedlicher Form und Größe) und die BIC-Werte für verschiedene GMMs.

und

$$\Sigma_2 = \begin{pmatrix} 2 & -2 \\ -2 & 3 \end{pmatrix}.$$

Abbildung 6.5 zeigt wieder die Daten und die BIC-Werte. Auch hier werden wieder zwei Komponenten erkannt, wir brauchen aber wegen der vollkommen unterschiedlichen Kovarianzmatrizen das volle Modell, das die Bezeichnung „VVV“ trägt. Wir erhalten:

```
Mclust VVV (ellipsoidal, varying volume, shape, and orientation)
model with 2 components:

 log.likelihood    n df       BIC      ICL
       -4061.03 1000 11 -8198.046 -8200.69
```

Die angegebenen 11 geschätzen Parameter sind die Mittelwerte (4 Freiheitsgrade), die Anteile der Komponenten (1 Freiheitsgrad) und 2 freie Kovarianzmatrizen (2·3 Freiheitsgrade, da diese symmetrisch sind). Auch hier liegen die Schätzer wieder in der Nähe der wahren Parameter:

```
> c$parameters$pro
[1] 0.5008032 0.4991968
> c$parameters$mean
         [,1]      [,2]
[1,] 3.008771 -1.936562
[2,] 3.952634 -1.074529
> c$parameters$variance$sigma
, , 1

          [,1]      [,2]
[1,] 1.8712029 0.9016049
[2,] 0.9016049 2.8935261

, , 2

          [,1]      [,2]
[1,]  2.024596 -2.055615
[2,] -2.055615  3.218936
```

◊

7. Finden und Erzeugen von Repräsentanten

In diesem abschließenden Kapitel sollen nun Methoden vorgestellt werden mit denen aus einer Menge von Messungen Repräsentanten für diese Menge ausgewählt werden können. Dafür sollen die im vorigen Kapitel vorgestellten Methoden zum Clustern von Daten Anwendung finden.

7.1. Auswählen von Repräsentanten

Um aus einer Menge von Messungen eine oder mehrere typische Messungen auszuwählen bietet sich der in Abschnitt 6.2 vorgestellte Partitioning-Around-Medoids (PAM) Algorithmus an. Dieser bestimmt ja neben den Clustern auch Medoide, die die Zentren der gefundenen Cluster darstellen. Man kann nun also für jede zu repräsentierende Gruppe in den Daten (wie zum Beispiel die Gruppe „Müllfahrzeuge“) mit Hilfe des Silhouettenkoeffizienten die optimale Anzahl an Clustern herausfinden, und die Medoide der gefundenen Cluster als Repräsentanten nutzen. Das wollen wir nun tun. Wir verwenden jeweils mit tf-idf gewichtete Feature-Vektoren welche die 3 Messkanäle „Fahrzeuggeschwindigkeit“, „Motordrehzahl“ und „Motordrehmoment“ (aneinander gehängt) beinhalten. Die SAX-Parameter sind wie gehabt eine Fensterlänge von 40, eine Wortlänge von $w = 4$ und ein Alphabet mit $a = 4$ Buchstaben. Die Vektoren sind also von der Dimension 768. Davon wollen wir die ersten 10 Hauptkomponenten (für jede Gruppe extra bestimmt) verwenden.

Ein Nachteil dieser Methode ist die —durch die große Länge der Messungen bedingte— schwierige Interpretation des Ergebnisses. Das Verfahren liefert zwar eine typische Messung pro Cluster, es ist aber nicht leicht in dieser Messung von meist mehreren tausend Sekunden interpretierbare Charakteristika zu identifizieren, die sie von ebenso langen Messungen in anderen Clustern unterscheidet.

Das Vorgehen soll dennoch an den zwei Gruppen „Müllfahrzeuge“ und „Autobahn / Bundesstraße“ gezeigt werden.

7.1.1. Müllfahrzeuge

Geclustert wurden hier die 13 Messungen der Müllfahrzeuge. Der Silhouettenkoeffizient legt ganz klar das Vorhandensein von 2 Clustern nahe:

k	2	3	4	5	6
Silhouettenkoeffizient	0.76	0.18	0.13	0.16	0.15

Es zeigt sich aber, dass bei 2 Clustern einer der beiden Cluster nur aus einem einzelnen Punkt besteht. Dies ist genau jene Messung, die auch schon beim hierarchischen Clustern (Abbildung 6.2) als einzige nicht in der Gruppe der anderen Müll-Messungen aufschien. Beim Betrachten dieser Messung fällt auf, dass die Motordrehzahl über weite Teile (>90%) der Messung 0 ist, also der Motor nicht gestartet war. Dementsprechend ist es auch nicht

verwunderlich, dass diese Messung nicht zu den anderen passt. Wir verwerfen diese Messung als Ausreißer und betrachten wieder die Silhouettenkoeffizienten auf den restlichen 12 Messungen für verschiedene k:

k	2	3	4	5	6
Silhouettenkoeffizient	0.19	0.14	0.18	0.17	0.14

Der Silhouettenkoeffizient ist bei $k = 2$ am größten, dort aber immer noch recht klein. Es könnte also auch sein, dass die Daten aus nur einem Cluster bestehen. Wir teilen dennoch die verbleibenden Punkte in 2 Cluster ein (Abbildung 7.1).

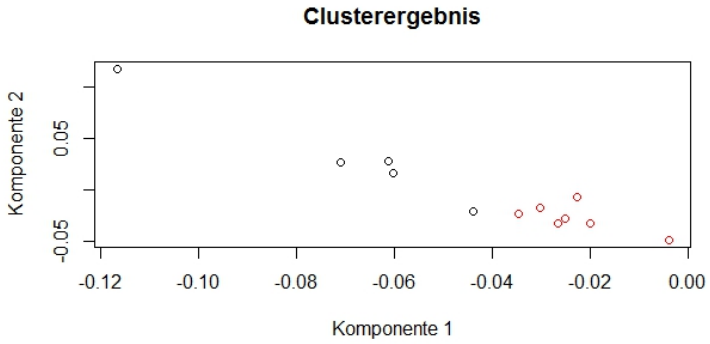


Abbildung 7.1.: Clusterzuordnung der Müllfahrzeugmessung, dargestellt über den ersten beiden Hauptkomponenten.

Beim Betrachten der zwei gefundenen Medoide fällt auf, dass beide Messungen sowohl den typischen Müllzyklus, als auch Abschnitte mit höherer Geschwindigkeit (bis ca 90 km/h) enthalten in denen die Fahrzeuge wohl zum nächsten Einsatzort gefahren sind. Die einzigen erkennbaren Unterschiede liegen in der Anzahl und Dauer dieser Phasen mit hoher Geschwindigkeit, sowie der erreichten Höchstgeschwindigkeit. Dies spiegelt sich auch in Plots der empirischen Verteilungsfunktionen der drei verwendeten Messkanäle wieder (Abbildung 7.2). Dort ist klar erkennbar, dass die grün gezeichnete Messung eine viel längere Zeit im Bereich von 20-60 km/h verbracht hat. Geschwindigkeiten über 60 km/h nehmen in den beiden Messungen in etwa den gleichen Anteil ein, auch wenn die Maximalgeschwindigkeit der grünen Messung mit 92.8 km/h etwas über jener der roten Messung (86.0 km/h)liegt. Auch die Verteilungen von Drehzahl und Drehmoment zeigen einen größeren Anteil der höheren Bereiche in der grünen Messung. Im Plot der Verteilung der Drehzahl ist außerdem zu erkennen, dass der Motor in der roten Messung länger nicht gestartet war als in der grünen, wodurch sich der Unterschied bei 0 Umdrehungen/Minute in den Verteilungsfunktionen erklärt.

7.1.2. Autobahn/Bundesstraße

Hier wurden die 13 Messungen von Fahrten auf Autobahn und Bundesstraße geclustert. Der Silhoettenkoeffizient für verschiedene k ergibt sich als:

Die Daten scheinen sich also wieder in 2 Gruppen zu gliedern. Der Silhouetten-Plot, in dem die einzelnen Silhouetten, sowie der mittlere Wert der Silhouetten pro Cluster

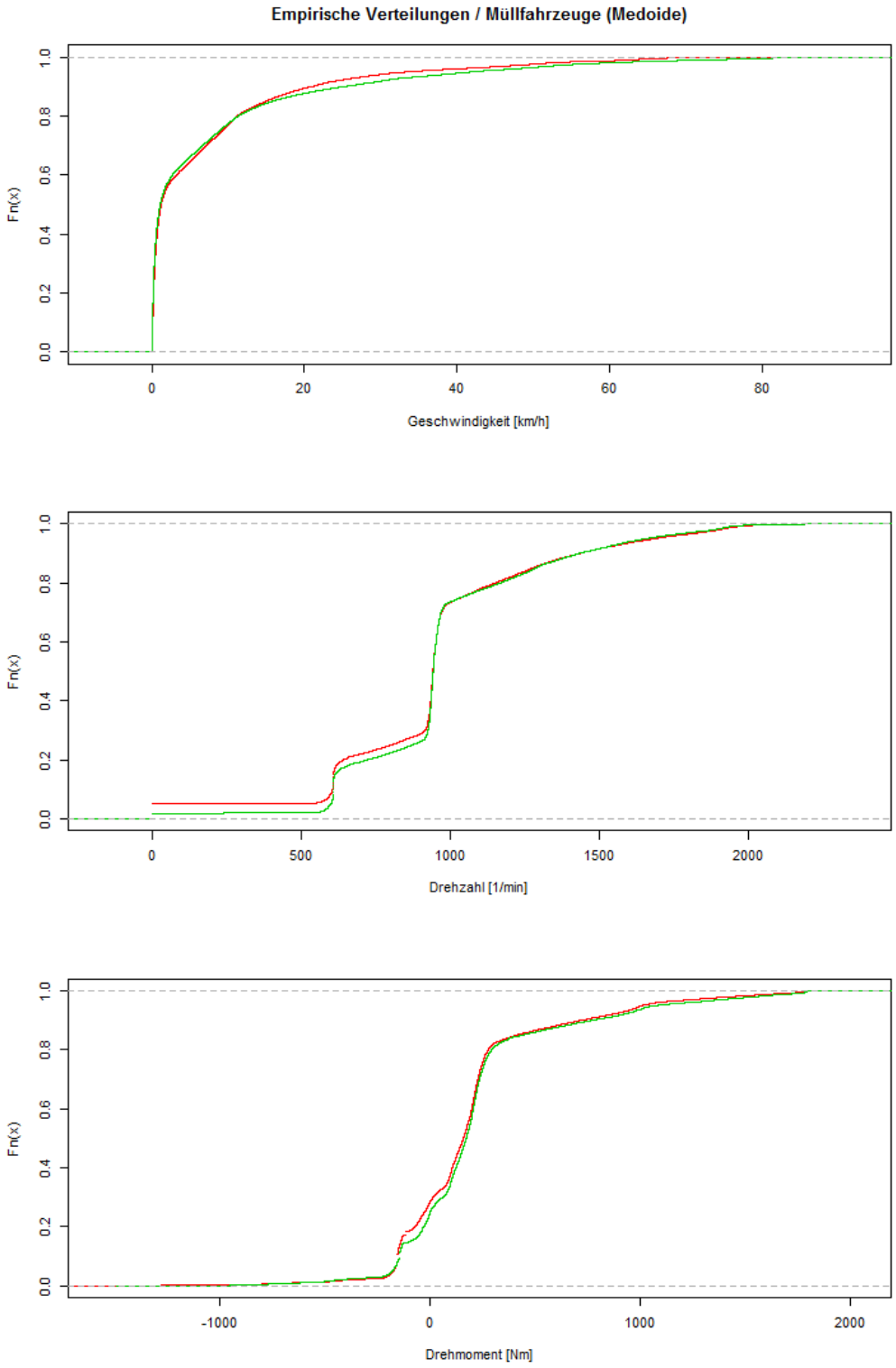


Abbildung 7.2.: Empirische Verteilungsfunktionen der 3 verwendeten Messkanäle der beiden Medoide der Müllfahrzeuge.

k	2	3	4	5	6
Silhouettenkoeffizient	0.59	0.54	0.49	0.44	0.40

angegeben sind, zeigt, dass der zweite Cluster eine viel größere Ausdehnung besitzt als der erste (Abbildung 7.3). Dennoch werden wir 2 Cluster verwenden.

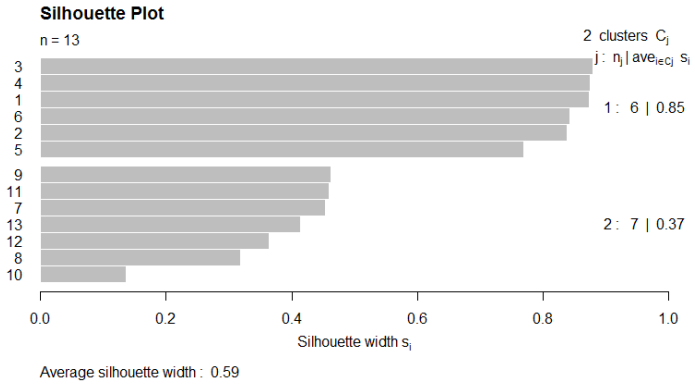


Abbildung 7.3.: Silhouetten-Plot der Aufteilung der Autobahn/Bundesstraße Daten in 2 Cluster.

In dieser Situation ist ein sehr großer Unterschied in den 2 Medoiden zu erkennen. Während Messung 1 (grün) über weite Strecken eine konstante Geschwindigkeit beibehält, variiert diese in Messung 2 (rot) viel stärker. Das ist sogar in einem Plot der gesamten Zeitreihe leicht zu erkennen (Abbildung 7.4).

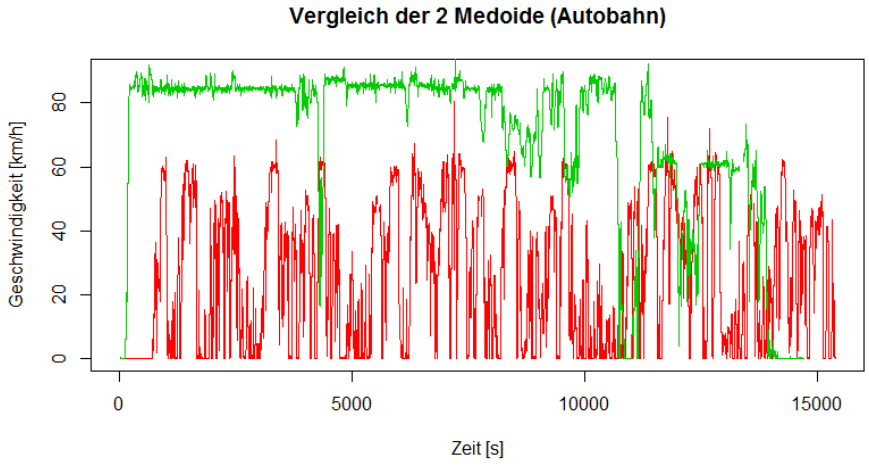


Abbildung 7.4.: Plot der Geschwindigkeiten der zwei als Medoide gewählten Messungen.

Natürlich liefern auch die empirischen Verteilungsfunktionen der 3 Messkanäle (Abbildung 7.5) sehr unterschiedliche Strukturen, wobei klar erkennbar ist, dass die grüne Messung einen viel höheren Zeitanteil mit hohen Geschwindigkeiten gefahren ist.

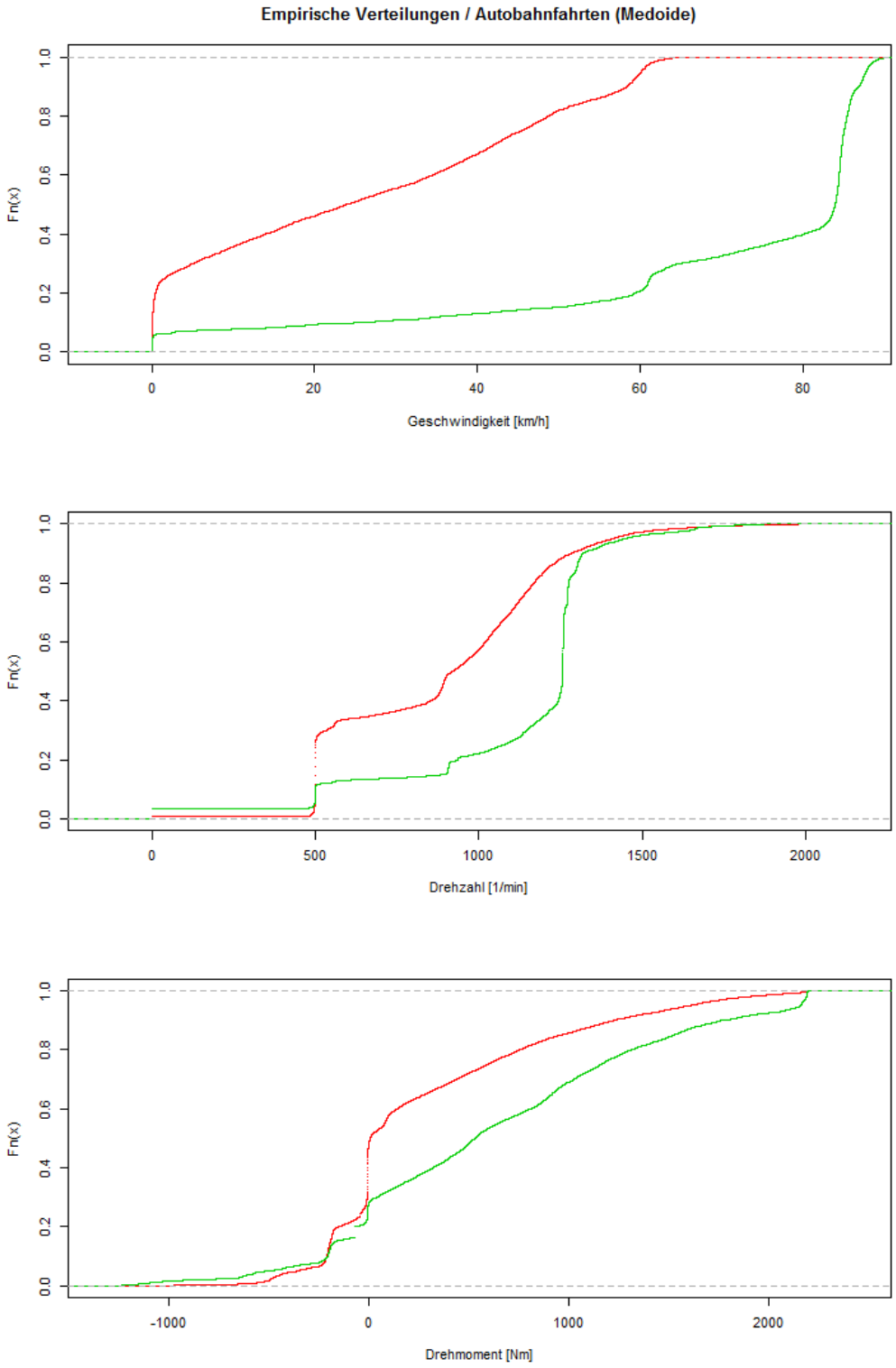


Abbildung 7.5.: Empirische Verteilungsfunktionen der 3 verwendeten Messkanäle der beiden Medoide der Autobahndaten.

7.2. Erzeugen von Repräsentanten

Zuletzt soll nun noch eine Methode zum Erzeugen von Repräsentanten aus Teilen der Originalmessungen vorgestellt werden. Zunächst wird die dazu verwendete Vorgangsweise beschrieben, danach die Ergebnisse.

Als erstes werden die vorhandenen Messungen in einzelne Teile gesplittet. Die Messungen werden an Stellen geteilt, an denen entweder die Fahrzeuggeschwindigkeit, die Motordrehzahl oder das Motordrehmoment für mindestens 5 Sekunden sehr klein sind. Genauer muss die Geschwindigkeit < 0.5 km/h, die Drehzahl < 100 U/min oder der Absolutbetrag des Drehmoments < 50 Nm sein. Von den entstehenden Stücken betrachten wir dann nur jene, deren Länge mindestens 40 Sekunden beträgt. Es zeigt sich, dass mit diesen Schwellwerten meist das Drehmoment eine Aufteilung bewirkt. Die Fahrzeuggeschwindigkeit ist nämlich —vermutlich durch Messungenauigkeiten— praktisch nie unter 1 km/h.

Als nächstes werden die Feature-Vektoren der Stücke erzeugt. Wir verwenden jene BOP-Repräsentation, in der die Feature-Vektoren der 3 Messkanäle aneinandergehängt werden. Die SAX-Parameter für jeden Kanal sind eine Fensterlänge von 40, eine Wortlänge von $w = 4$ und es werden $a = 4$ Symbole verwendet. Die Fenster werden nicht normalisiert und tf-idf wird nicht verwendet. Die Breakpoints wurden im Gegensatz zu vorher nicht fix gewählt, sondern für jede Gruppe (Autobahnfahrten, Müllfahrzeuge, etc.) extra berechnet, in dem die empirischen Quartile der Punkte der Messungen in jeder Gruppe bestimmt werden.

Nun werden die Feature-Vektoren der Teile aus jeweils einer Gruppe durch ein GMM modelliert und damit geculstert. Da die Dimension der Vektoren mit 768 jedoch viel zu groß ist um ein sinnvolles Modell zu finden, werden nur die ersten 15 Hauptkomponenten der Vektoren (wieder pro Gruppe berechnet) verwendet. Die so gefundenen Cluster können dann analysiert werden. Zum Erzeugen eines Referenzzyklus kann man dann Teile aus den verschiedenen Clustern zusammensetzen um die Aspekte der Messungen die sie repräsentieren in den Referenzzyklus einzubringen. Dies soll nun an den verschiedenen Gruppen präsentiert werden:

7.2.1. Müllfahrzeuge

Beim Splitten der Messungen der Müllfahrzeuge ergaben sich 1779 Stücke mit einer durchschnittlichen Länge von 121 Sekunden. Der Plot der BIC-Werte für die verschiedenen Modelle zeigt klar, dass ein Modell mit vollen Kovarianzmatrizen notwendig ist.

Obwohl der BIC-Wert für 8 Komponenten etwas größer als jener für 7 Komponenten ist, zeigt es sich, dass 7 Komponenten ausreichen um die verschiedenen Strukturen abzubilden. Nachfolgend ist eine Beschreibung der einzelnen Cluster über die Fahrzeuggeschwindigkeit angegeben:

Cluster 1:
Im ersten Cluster befinden sich 191 Teile. In diesen werden mittlere Geschwindigkeiten von 40-60 km/h erreicht, die in manchen Stücken länger gehalten, meist aber nur kurz erreicht werden.

Cluster 2:
Im zweiten Cluster befinden sich 218 Teile. Diese beinhalten anscheinend Fahrten der

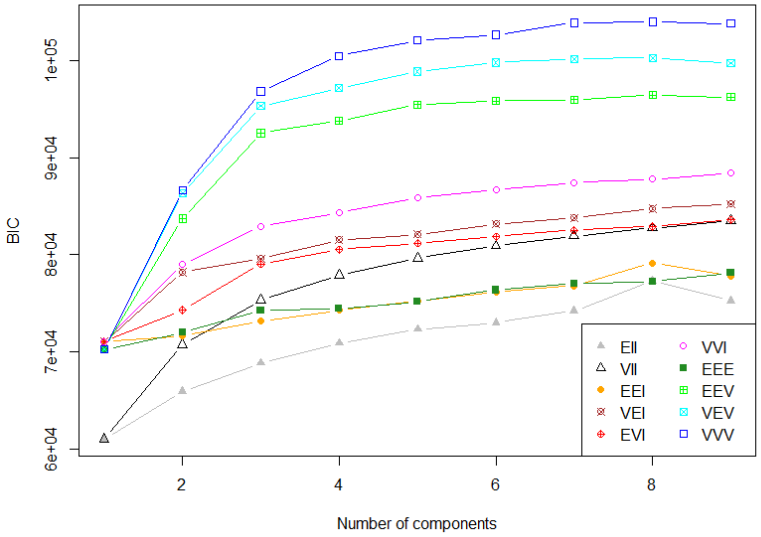


Abbildung 7.6.: BIC-Werte für verschiedene Modelle für die Feature-Vektoren der Teile der Müllfahrzeug-Messungen

Müllfahrzeuge zum nächsten Einsatzort, das heißt, die Geschwindigkeit der Fahrzeuge in den Stücken ist relativ konstant, oder ändert sich nur langsam. Die mittlere Geschwindigkeit beträgt meist 40-80 km/h, liegt jedoch in manchen Teilen auch darunter. In den Messungen in diesem Cluster stoppen die Fahrzeuge nur selten oder gar nicht, es handelt sich also vermutlich um Überlandfahrten. Beispiele von Messungen dieses Clusters sind in Anhang C.1 in Abbildung C.1 dargestellt.

Cluster 3:
Im dritten Cluster befinden sich 341 Teile. Dieser Cluster scheint Teile von Fahrten in der Stadt zu enthalten. Die Fahrzeuge stoppen relativ häufig und fahren dann wieder etwa 10-30 Sekunden lang.

Cluster 4:
Der vierte Cluster ist mit 345 Teilen in etwa gleich groß wie der dritte. Er enthält Teile in denen das typische Geschwindigkeitsprofil eines Müllwagens zu sehen ist, also kurzes Anfahren, wobei meist nicht mehr als 10-15 km/h erreicht werden, und direkt darauffolgendes Stoppen für 15-20 Sekunden. Die Messungen in diesem Cluster sind eher kurz, die meisten sind etwa 60-80 Sekunden lang. Beispiele von Messungen dieses Clusters sind in Anhang C.1 in Abbildung C.2 dargestellt.

Cluster 5:
Cluster 5 ist mit 434 Teilen mit Abstand der größte der 7 Cluster. Die Teile der Messungen in diesem Cluster sind von der Struktur her ähnlich wie jene in Cluster 4, enthalten also auch den Müllzyklus. Allerdings ist die gefahrene Strecke zwischen den einzelnen

Stopps geringer als in Cluster 4, entsprechend werden auch nur kleinere Geschwindigkeiten erreicht. Die Dauer der Stopps ist aber in etwa gleich. Außerdem sind die Stücke länger als jene im vierten Cluster, viele dauern über 200 Sekunden.

Cluster 6:
In Cluster 6 befinden sich 204 Teile. Diese sind durch lange (im Verhältnis zur Länge der Teile) Stopps, oder lange Fahrten mit sehr langsamer Geschwindigkeit unter 5 km/h charakterisiert. Es enthält trotzdem fast jedes Stück eine kurze Phase in der höhere Geschwindigkeiten bis ca 20 km/h erreicht werden.

Cluster 7:
Der 7. Cluster beinhaltet nur 46 Teile und ist damit der kleinste der gefundenen Cluster. Dieser Cluster enthält Teile mit Stoppphasen oder sehr kleiner Geschwindigkeit. Im Gegensatz zu Cluster 6 gibt es nur ganz wenige Teile, in denen Geschwindigkeiten über 5 km/h erreicht werden. Beispiele von Messungen dieses Clusters sind in Anhang C.1 in Abbildung C.3 dargestellt.

7.2.2. Autobahn / Bundesstraße

Aus den Messungen von Fahrten auf Autobahn und Bundesstraße wurden 672 Teile mit einer mittleren Länge von 201 Sekunden erzeugt. Die BIC-Werte für verschiedene Modelle (Abbildung 7.7) legen ein Modell mit vollen Kovarianzmatrizen und 5 Komponenten nahe.

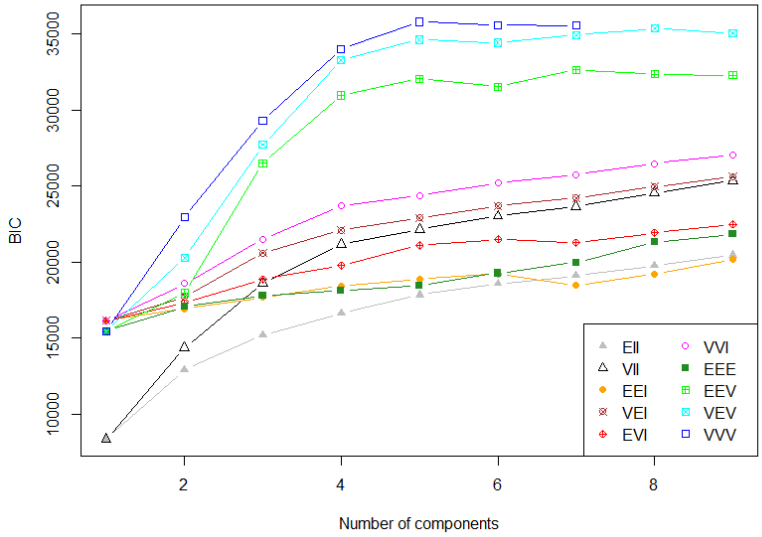


Abbildung 7.7.: BIC-Werte für verschiedene GMMs für die Teile aus Messungen auf Autobahn und Bundesstraße.

Beim Betrachten der Cluster zeigt sich, dass sich diese vor allem in den gefahrenen Geschwindigkeiten unterscheiden.

Cluster 1:
Cluster 1 enthält 132 Teile die sehr langsame Fahrten mit Geschwindigkeiten unter 30 km/h beinhalten. Hin und wieder kommen auch Stopps vor.

Cluster 2:
Cluster 2 enthält 147 Teile. Hier tauchen großteils Geschwindigkeiten von 40-60 km/h auf, wobei die Geschwindigkeit recht stark variiert.

Cluster 3:
Cluster 3 ist mit 174 Teilen der größte der 5 Cluster. Die gefahrene Geschwindkeit ist höher als in Cluster 1, übersteigt 50 km/h jedoch selten.

Cluster 4:
Der 4. Cluster enthält nur 76 Teile, und ist damit der kleinste. Er enthält Fahrten mit sehr konstanter Geschwindigkeit in der kaum Schwankungen zu erkennen sind. Interessanterweise bewegen sich diese konstanten Phasen jedoch auf komplett unterschiedlichen Niveaus, von 10-90 km/h. Beispiele von Messungen dieses Clusters sind in Anhang C.2 in Abbildung C.4 dargestellt.

Cluster 5:
Der 5. und letzte Cluster enthält 143 Teile mit ebenfalls relativ konstante Phasen, die Geschwindigkeit in diesen variiert allerdings etwas stärker als im 4. Cluster. Das Geschwindigkeitsniveau ist durchwegs über 80 km/h und damit hoch. Dieser Cluster enthält die typischen Autobahnfahrten von LKWs. Beispiele von Messungen dieses Cluster sind in Anhang C.2 in Abbildung C.5 dargestellt.

7.2.3. Verteilerverkehr

Aus den Messungen im Verteilerverkehr ergaben sich 285 Teile mit einer mittleren Länge von 321 Sekunden. Der Vergleich der BIC-Werte in Abbildung 7.8 legen ein Modell mit 8 Komponenten nahe, in dem die Achsen Kovarianzellipsen in jeder Komponente in die gleichen Richtungen zeigen. Beim Betrachten der Teile in den Clustern können jedoch keine für den jeweiligen Cluster charakteristischen Eigenschaften festgestellt werden, die ihn von anderen Clustern unterscheiden. Hier scheint die Methode also nicht sehr gut zu funktionieren.

7.2.4. Stadtverkehr

Die Messungen im Stadtverkehr ergaben 646 Stücke mit einer mittleren Länge von 191 Sekunden. Die Modellselektion über BIC liefert ein Modell mit vollen Kovarianzmatrizen und 5 Komponenten (Abbildung 7.9). Es zeigt sich, dass sich mit diesem Modell sehr unterschiedlich große Cluster ergeben. So beinhalten etwa die größten beiden Cluster bereits 524 der 646 Stücke.

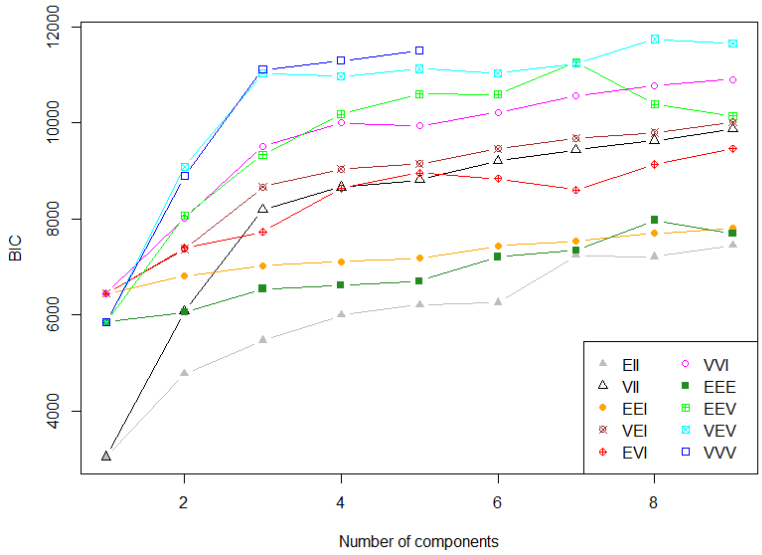


Abbildung 7.8.: BIC-Werte für Modelle für Teile von Messungen im Verteilerverkehr.

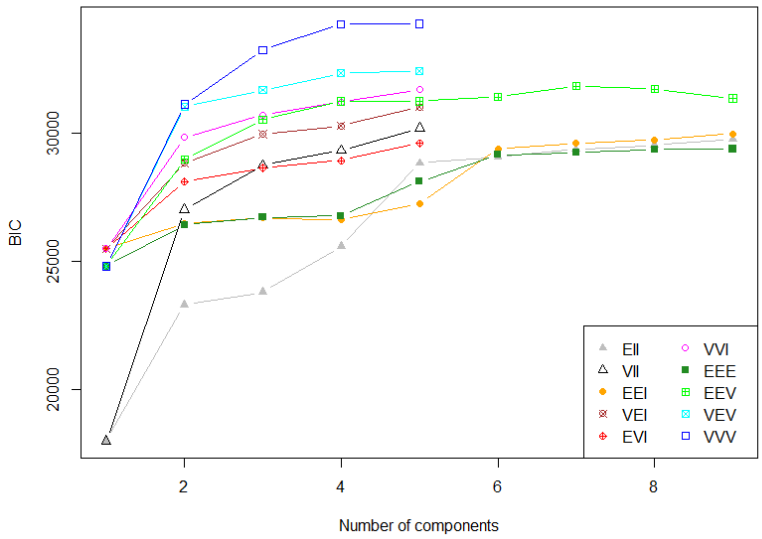


Abbildung 7.9.: BIC Werte für GMMs für Teile von Messungen aus dem Stadtverkehr.

Es zeigten sich die folgenden Charakteristika der Cluster:

Cluster 1:
Cluster 1 enthält nur 24 Stücke, in denen die gleiche Geschwindigkeit sehr konstant über jeweils das ganze Stück gehalten wird. Diese (im Stück konstante) Geschwindigkeit ist jedoch bei verschiedenen Stücken sehr unterschiedlich, aber immer unter 80 km/h. Damit ähneln die Messungen in diesem Cluster jenen in Cluster 4 der Autobahndaten.

Cluster 2:
Dieser Cluster ist mit 292 Teilen der größte der 5 Cluster. Er enthält Stücke in denen die Geschwindigkeit relativ stark variiert und im Mittel etwa 40 km/h beträgt. Es könnte sich um Straßen mit Ampeln zu handeln an denen zum Beispiel wegen eines Rückstaus abgebremst werden muss (daher die Variationen in der Geschwindigkeit). Immer wieder kommen die Fahrzeuge in den Messungen auch kurz ganz zum Stillstand. Beispiele von Messungen dieses Cluster sind in Anhang C.3 in Abbildung C.6 dargestellt.

Cluster 3:
Der 3. Cluster enthält 232 Beobachtungen, und ist damit der zweitgrößte. Die Geschwindigkeiten der Messungen in diesem Cluster variieren in den Stücken weniger stark als in Cluster 2 und die Durchschnittsgeschwindigkeit ist mit etwa 30 km/h geringer als dort. Stopps kommen im Gegensatz zu Cluster 2 in diesen Teilen kaum vor.

Cluster 4:
Cluster 4 enthält 77 Teile. Dieses haben zum Großteil eine von 2 Strukturen: Entweder sieht man zunächst einen langsamen Anstieg der Geschwindigkeit, die gegebenenfalls dann auf einem Wert konstant gehalten wird, oder das Stück beginnt mit einem konstanten Teil, der dann in einen Teil in dem die Geschwindigkeit langsam abfällt übergeht. Beispiele von Messungen dieses Cluster sind in Anhang C.3 in Abbildung C.7 dargestellt.

Cluster 5:
Cluster 5 ist mit 21 Beobachtungen der kleinste Cluster. Er enthält Stücke mit Fahrten mit kleinen Geschwindigkeiten unter 25 km/h.

7.2.5. Passstrecke

Für die Messungen auf einer Passstrecke ergaben sich 453 Teile mit einer großen mittleren Länge von 481 Sekunden. Der Plot der BIC-Werte (Abbildung 7.10) legt ein Modell mit vollen Kovarianzmatrizen und 4 Komponenten nahe.

Beim Betrachten der gefundenen Cluster zeigte sich, dass nur 2 der 4 Cluster gut charakterisiert werden können:

Cluster 1:
Im ersten Cluster befinden sich 147 Teile, in denen meistens eine Geschwindigkeit relativ konstant gehalten wird. Meist ist diese Geschwindigkeit etwa bei 60 km/h, es gibt jedoch auch einige Teile in denen sie größer oder kleiner ist. Insgesamt ähneln die Teile in diesem Cluster jenen in Cluster 4 der Autobahndaten und Cluster 1 der Stadtdaten.

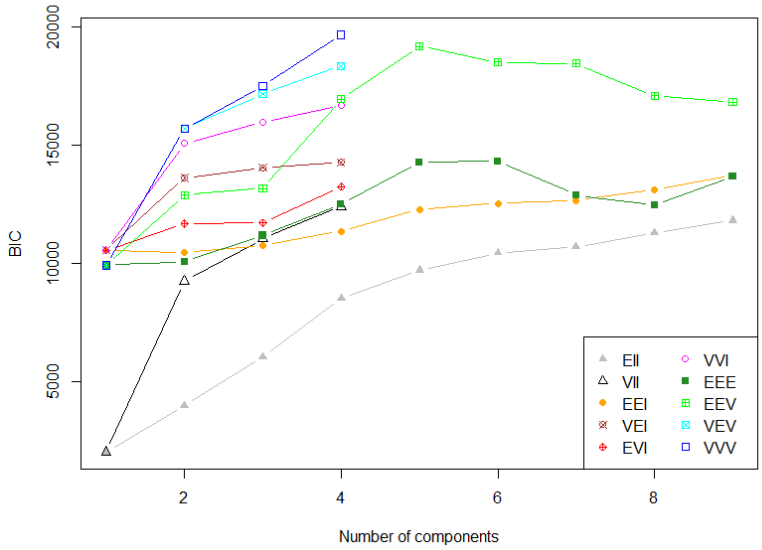


Abbildung 7.10.: BIC-Werte für GMMs für die Teile der Messungen auf einer Passstrecke.

Cluster 2:
Cluster 2 enthält 118 Messungen. Viele davon sind durch einen relativ konstanten Anstieg oder Abfall der Geschwindigkeit charakterisiert, aber es gibt auch einige die eine ganz andere Struktur haben (sowohl sehr große als auch sehr kleine Variation). Insgesamt ist es schwer, diesen Cluster zu charakterisieren.

Cluster 3:
Der dritte Cluster enthält nur 48 Stücke und ist damit der kleinste der 4 gefundenen. Die Messungen in diesem Cluster enthalten fast ausnahmslos sehr hohe Geschwindigkeiten von 80-100 km/h. Diese werden jeweils über das gesamte Stück sehr konstant gehalten.

Cluster 4:
In Cluster 4 befinden sich 140 Teile. Auch diese sind eher schwer zu charakterisieren. Eine Gemeinsamkeit scheint zu sein, dass die Spannweite der Geschwindigkeitswerte bei allen Stücken recht hoch ist. Im Mittel liegt die Geschwindigkeit bei etwa 60 km/h.

8. Überblick über die gewonnenen Erkenntnisse

Zur Erstellung von Referenzzyklen ist es notwendig, die vorhandenen Messungen oder Teile davon so zu repräsentieren, dass ähnliche Messungen auch ähnliche Repräsentationen haben. In der Arbeit wurden mehrere solcher Repräsentationen untersucht.

Im ersten Teil wurde versucht, die Messungen durch die in ihnen vorkommenden Muster zu charakterisieren. Zu diesem Zweck wurde das Distanzmaß *Dynamic Time Warping* (DTW) vorgestellt, das zeitliche Verzerrungen in verschiedenen Vorkommen des Musters ausgleichen kann, indem es mit dynamischer Optimierung eine optimale Zuordnung der Punkte in beiden Stücke zueinander findet. Danach wurde ein probabilistischer Algorithmus für die Mustersuche präsentiert und sowohl mit DTW als auch mit anderen Distanzmaßen getestet. Es stellte sich heraus, dass der Algorithmus zwar auf künstlich erzeugten oder sehr einfachen und kurzen realen Daten gute Ergebnisse liefert, aber die Performance auf den, in der Praxis auftauchenden, Daten sehr stark nachlässt. Dies liegt daran, dass der Algorithmus immer nur einige zufällig gewählte Teilstücke der Messungen miteinander vergleicht. Aufgrund der großen Länge der Messungen und dem relativ seltenen Auftreten der einzelnen Muster müsste man deshalb die Anzahl dieser zufällig gewählten Teilstücke sehr groß wählen, was jedoch aufgrund der Zeitkomplexität des Algorithmus schnell zu Laufzeiten von mehreren Jahren führen würde. Damit wurde diese Methode zur Charakterisierung der Messungen verworfen.

Im zweiten Teil wurde die *Symbolic Aggregate approXimation* (SAX) für Zeitreihen präsentiert, mit der eine Zeitreihe mit einem Wort fixer Länge über einem endlichen Alphabet approximiert werden kann. Darauf aufbauend wurde die *bag-of-patterns* (BOP) Repräsentation für Zeitreihen vorgestellt. Bei dieser Methode werden Teilstücke einer Zeitreihe mit einem *sliding window* extrahiert und das SAX-Wort dieses Teilstücks bestimmt. Die Repräsentation der gesamten Zeitreihe ergibt sich dann als Vektor der Häufigkeiten der vorkommenden SAX-Wörter. Diese berücksichtigt also nur die Häufigkeit der in der Zeitreihe vorkommenden Strukturen, aber nicht deren Reihenfolge. Mit diesen *Feature-Vektoren* der Häufigkeiten wurden dann *Support-Vector-Machines* (SVMs) trainiert um die Nützlichkeit der BOP-Repräsentation zu beurteilen. Dazu wurden neue Messungen, deren Gruppenzugehörigkeit jedoch bekannt war, durch die trainierte SVM klassifiziert, wobei bereits gute Ergebnisse erzielt wurden. Es wurden dann verschiedene Varianten von SAX vorgestellt und deren Performance mit Hilfe von SVMs bewertet, darunter auch drei Möglichkeiten mehrere Messkanäle simultan in die Repräsentation einfließen zu lassen. Die beste Performance zeigte sich dabei, wenn die verwendeten Messkanäle einzeln durch relativ kurze SAX-Worte repräsentiert wurden und die entstehenden Feature-Vektoren durch Aneinanderhängen zu einem langen Feature-Vektor kombiniert wurden (Abschnitt 5.1).

Im dritten Teil wurden drei Methoden zum Clustern von Daten vorgestellt. Mit Hilfe dieser Clustermethoden und der im vorigen Teil gefundenen Repräsentation der Messungen wurden dann zwei Möglichkeiten zur Erstellung von Referenzzyklen gezeigt. Dazu wurden

die, den Messungen entsprechenden Feature-Vektoren als Punkte im Raum betrachtet und geclustert. Die erste Methode wählt Messungen als Referenzzyklus aus, deren Feature-Vektoren in der Mitte von Clustern liegen. Diese werden mit dem *Partitioning Around Medoids* (PAM) Verfahren gefunden. Es zeigte sich jedoch, dass die mit dieser Methode gefundenen Cluster, beziehungsweise deren Mittelpunkte, nur schwer interpretierbar sind, da ja immer ganze Messungen betrachtet werden. Diese sind jedoch sehr lang und es lassen sich mit freiem Auge kaum für einen Cluster typische Charakteristika erkennen.

In der zweiten vorgestellten Methode zum Erstellen von Referenzzyklen wurden die Messungen deshalb in kleinere Teile zerteilt. Die Teilungspunkte ergaben sich aus den gemessenen Werten, waren diese über längere Zeit klein, so wurde geteilt. Diese kurzen Teilstücke wurden nun wieder durch BOP repräsentiert. Die entstehenden Datenpunkte wurden mit Hilfe eines gefitteten *Gaussian Mixture Models* (GMM) geclustert. Es zeigte sich dabei, dass fast alle entstehenden Cluster gut interpretierbar waren, das heißt, es konnten viele Cluster typischen Fahrsituationen zugeordnet werden. Auch optisch hatten die Teilstücke in den einzelnen Clustern jeweils sehr ähnliche Strukturen. Referenzzyklen für bestimmte Situationen können nun erzeugt werden, in dem Teilstücke aus einem oder mehreren Clustern wieder zu einem längeren Zyklus kombiniert werden.

Literaturverzeichnis

Catalano, J., Armstrong, T. und Oates, T. Discovering patterns in real-valued time series. In Fürnkranz, J., Scheffer, T. und Spiliopoulou, M., editors, *Knowledge Discovery in Databases: PKDD 2006*, Volume 4213 of *Lecture Notes in Computer Science*, pages 462–469. Springer Berlin Heidelberg, 2006.

Cleveland, W. S. Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 74(368):829–836, 1979.

Dempster, A. P., Laird, N. M. und Rubin, D. B. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, Series B (methodological)*, 39(1):1–38, 1977.

Fraley, C. und Raftery, A. E. Model-based clustering, discriminant analysis and density estimation. *Journal of the American Statistical Association*, 97:611–631, 2002.

Giorgino, T. Computing and visualizing dynamic time warping alignments in R: The dtw package. *Journal of Statistical Software*, 31(7):1–24, 2009.

Keogh, E. und Kasetty, S. On the need for time series data mining benchmarks: A survey and empirical demonstration. *Data Mining and Knowledge Discovery*, 7(4):349–371, October 2003.

Kuhn, H. W. und Tucker, A. W. Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492, Berkeley, Calif., 1951. University of California Press.

Lin, J. und Li, Y. Finding structural similarity in time series data using bag-of-patterns representation. In Winslett, M., editor, *Scientific and Statistical Database Management*, Volume 5566 of *Lecture Notes in Computer Science*, pages 461–477. Springer Berlin Heidelberg, 2009.

Lin, J., Keogh, E., Wei, L. und Lonardi, S. Experiencing sax: a novel symbolic representation of time series. *Data Mining and Knowledge Discovery*, 15(2):107–144, 2007.

Maechler, M., Rousseeuw, P., Struyf, A., Hubert, M. und Hornik, K. *cluster: Cluster Analysis Basics and Extensions*, 2014. R package version 1.15.2.

Manning, C. D., Raghavan, P. und Schuetze, H. Scoring, term weighting, and the vector space model. In *Introduction to Information Retrieval*, pages 100–123. Cambridge University Press, 2008. Cambridge Books Online.

Rajaraman, A. und Ullman, J. D. Data mining. In *Mining of Massive Datasets*, pages 1–17. Cambridge University Press, 2011. Cambridge Books Online.

Rousseeuw, P. und Kaufman, L. Clustering by means of medoids. In *Statistical Data Analysis Based on the L1 Norm and Related Methods*, pages 405–416. North-Holland, 1987.

Rousseeuw, P. J. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

Salton, G., Wong, A. und Yang, C. S. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, November 1975.

Schwarz, G. Estimating the dimension of a model. *The Annals of Statistics*, 6(2):461–464, 1978.

Tanaka, Y., Iwamoto, K. und Uehara, K. Discovery of time-series motif from multi-dimensional data based on mdl principle. *Machine Learning*, 58(2-3):269–300, 2005.

Vapnik, V. N. *Statistical Learning Theory*, Volume 2. Wiley New York, 1998.

Ward Jr, J. H. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, 1963.

Wendemuth, A. *Grundlagen der stochastischen Sprachverarbeitung*. Oldenbourg, 2004.

A. Verwendete Parameter und gefundene Muster der einzelnen Experimente

Die in diesem Anhang gezeigten Muster entsprechen einer willkürlich getroffenen Auswahl aus einem Durchlauf des Algorithmus. Diese sollen als Beispiele für die gefundenen Muster dienen. In jeder Situation wurde vor dem Starten des Algorithmus der Seed des Zufallszahlen-Generators auf denselben Wert festgelegt.

A.1. Künstliche Daten

Die gefundenen Muster stellen allesamt (Teile von) Instanzen des tatsächlich vorhandenen Musters dar. Es wurden keine Stücke die das künstliche Muster nicht enthalten als Muster erkannt. Die erkannten Muster enthalten kaum zusätzliche, nicht zum Muster gehörende Punkte am Anfang oder Ende. Eine Auswahl der Muster ist unten dargestellt.

Verwendete Parameter	**Wert**
window size w	50
subwindow size $\bar{w}$	5
k	5
gesampelte Fenster für Kandidatenmenge	15
gesampelte Fenster für Vergleichsmenge / Iteration	10

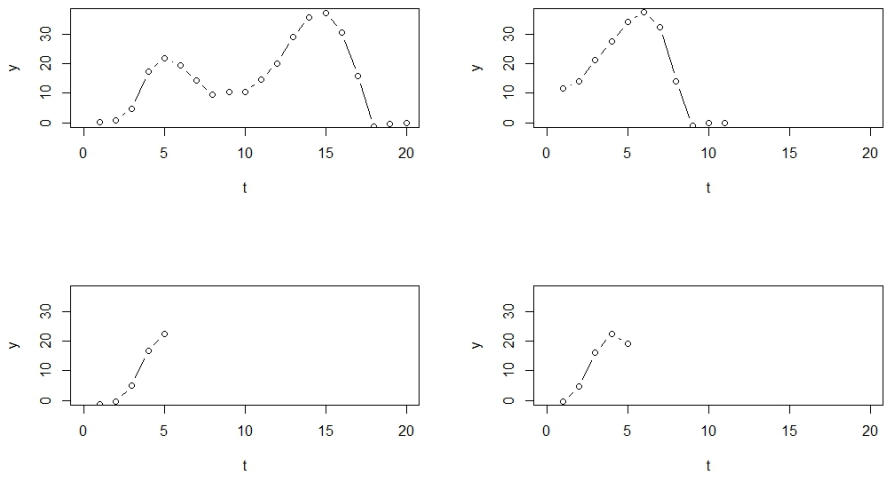


Abbildung A.1.: Mustererkennung in der künstlichen Zeitreihe: verwendete Parameter und gefundene Muster (Auswahl)

A.2. Einfache reale Daten

A.2.1. Distanzmaß: DTW

Mit DTW als Distanzmaß wurden in den einfachen realen Daten (Geschwindigkeit eines Müllfahrzeugs) vor allem Muster gefunden in denen die Fahrzeuggeschwindigkeit sehr klein war, wie etwa Stehphasen. Der typische „Müllzyklus“, also das Anfahren und das kurz darauffolgende Stoppen wurde nur in sehr kurzen Varianten (in denen das Fahrzeug nur eine kleine Maximalgeschwindigkeit erreicht) erkannt. Das könnte daran liegen, dass bei längeren Zyklen mit höherer Geschwindigkeit mehr Möglichkeiten und Zeit zur Variation für den Fahrer besteht, während sehr kurze Abschnitte immer gleich gefahren werden (müssen).

Verwendete Parameter	**Wert**
window size w	40
subwindow size $\bar{w}$	5
k	5
gesampelte Fenster für Kandidatenmenge	15
gesampelte Fenster für Vergleichsmenge / Iteration	10

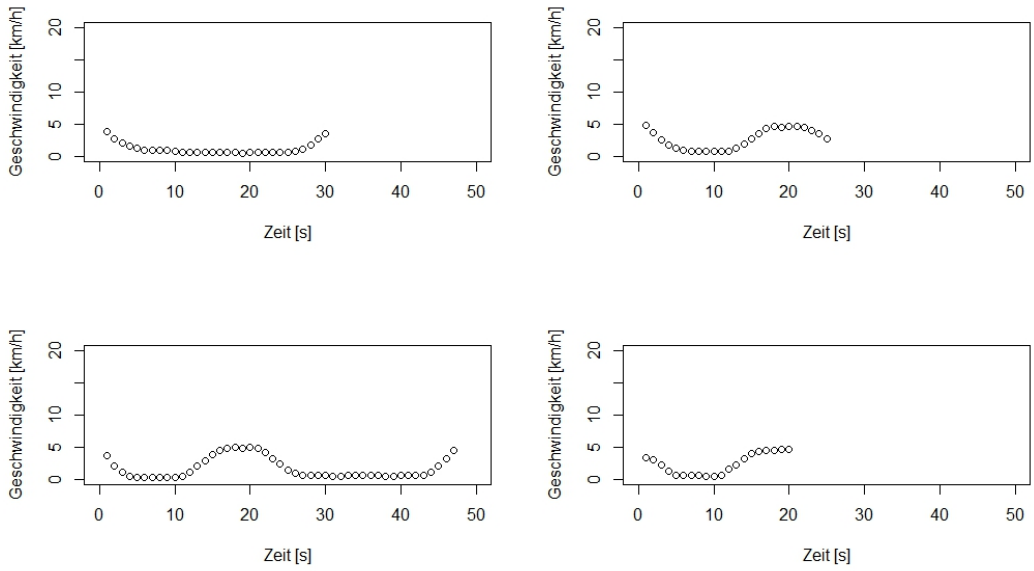


Abbildung A.2.: Mustersuche in der Müllfahrzeug-Zeitreihe: Verwendete Parameter und mit DTW gefundene Muster (Auswahl)

A.2.2. Distanzmaß: Korrelationskoeffizient

Wendet man den Algorithmus auf die einfachen realen Daten (Geschwindigkeit eines Müllfahrzeugs) mit dem Korrelationskoeffizienten als Distanzmaß an, so werden einige Muster gefunden. Diese entsprechen dem Anfahren des Müllfahrzeuges bis zum bald dar-

auffolgenden Stoppen. Im Gegensatz zu den mit DTW gefundenen Mustern werden hier auch längere Varianten dieser „Müllzyklen“ (oder Teile davon) erkannt. Stehzeiten des Fahrzeugs werden jedoch nicht erkannt, obwohl man diese durchaus auch als Muster interpretieren könnte. Das liegt wohl daran, dass bei Geschwindigkeit 0 die Messwerte aus weißem Rauschen bestehen, also Korrelation 0 zwischen solchen Stücken besteht.

Verwendete Parameter	**Wert**
window size w	40
subwindow size $\bar{w}$	5
k	5
gesampelte Fenster für Kandidatenmenge	15
gesampelte Fenster für Vergleichsmenge / Iteration	10

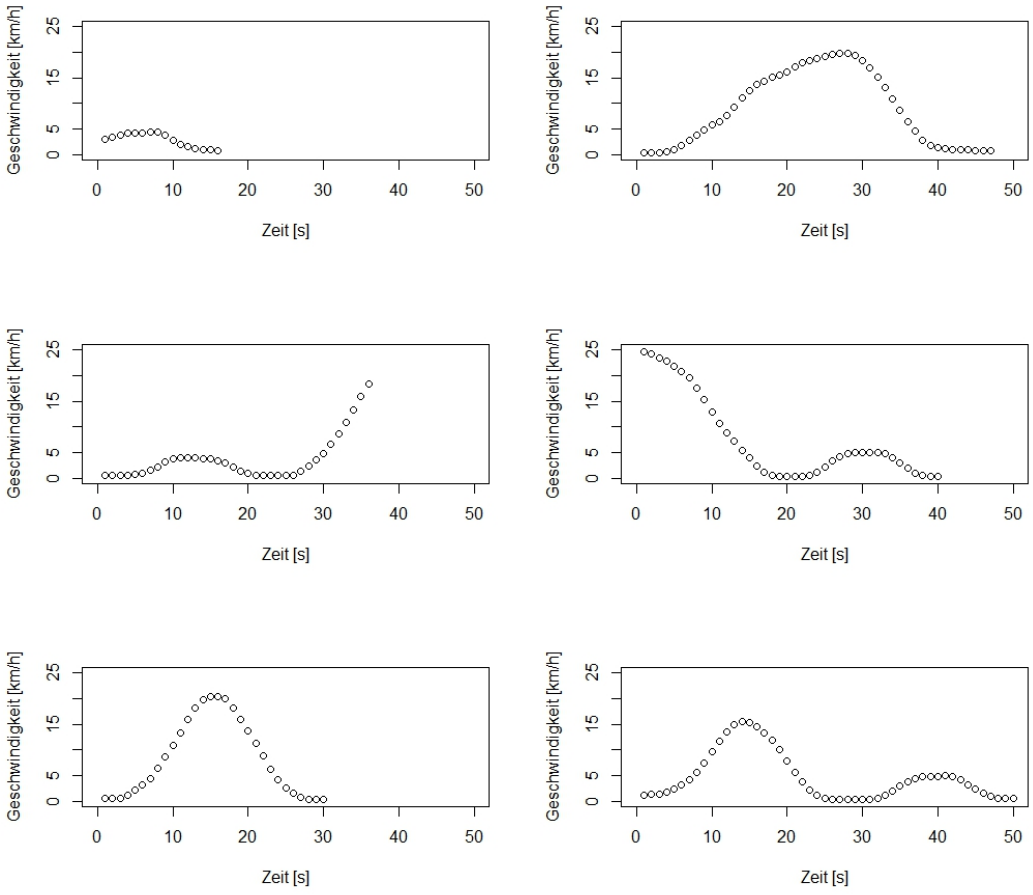


Abbildung A.3.: Mustersuche in der Müllfahrzeug-Zeitreihe: Verwendete Parameter und mit dem Korrelationskoeffizienten gefundene Muster (Auswahl)

A.3. Reale Daten

A.3.1. Distanzmaß DTW

Wie schon bei den einfachen realen Daten, hat sich auch bei den komplizierteren realen Daten gezeigt, dass mit dem Distanzmaß DTW vor allem Phasen in denen die Fahrzeuggeschwindigkeit konstant bleibt, beziehungsweise nur leicht steigt oder fällt als Muster erkannt werden. Dadurch, dass das betrachtete Fahrzeug aber auch auf der Autobahn unterwegs war werden solche „konstanten Muster“ auch im höheren Geschwindigkeitsbereich gefunden. Muster die man bestimmten, komplexeren, Fahrmanövern zuordnen könnte werden aber nicht gefunden.

Parameter	**Wert**
window size w	100
subwindow size $\bar{w}$	8
k	3
gesampelte Fenster für Kandidatenmenge	10
gesampelte Fenster für Vergleichsmenge / Iteration	10

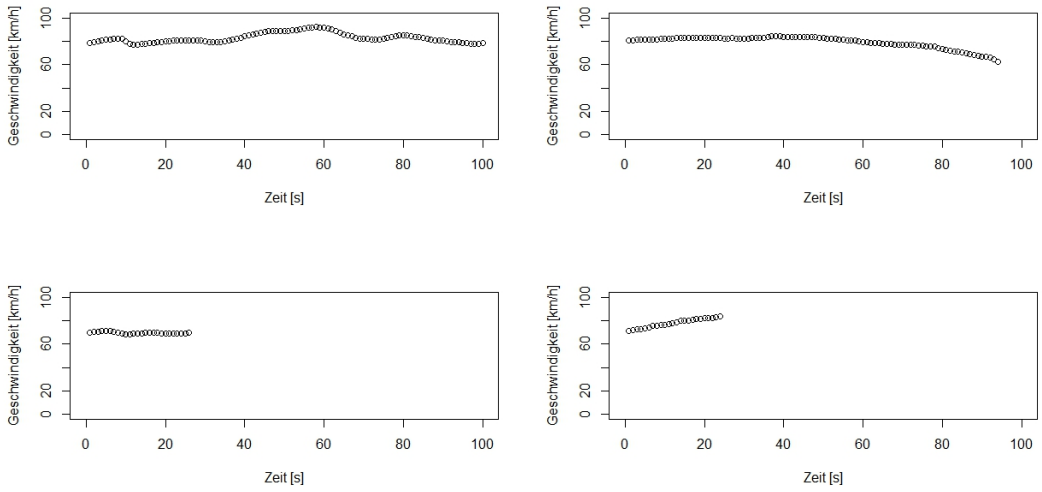


Abbildung A.4.: Passende Parameter für DTW und damit gefundene Muster (Auswahl)

A.3.2. Distanzmaß 1-Norm

Mit der 1-Norm als Distanzmaß werden vom Algorithmus längere und komplexere Muster gefunden als mit DTW. Allerdings scheinen alle gefundenen Muster komplett verschieden zu sein, es ist also fraglich in wie weit die gefundenen Stücke Muster darstellen, wenn diese in der Messung nur recht selten vorkommen. Auch eine Zuordnung zu bestimmten Fahrmanövern ist großteils nicht möglich. Nur in manchen Fällen sind bestimmte Charakteristika erkennbar. So könnte das rechte obere Muster in Abbildung A.5 von einem Stück Bundesstraße stammen in dem immer wieder Zonen mit einer Geschwindigkeitsbegren-

zung von 50 km/h auftauchen (etwa Ortsgebiet o.Ä.). Informationen dieser Art könnten durchaus verwendet werden um die Messung zu charakterisieren.

Parameter	**Wert**
window size w	500
subwindow size $\bar{w}$	8
k	25
gesampelte Fenster für Kandidatenmenge	10
gesampelte Fenster für Vergleichsmenge / Iteration	10

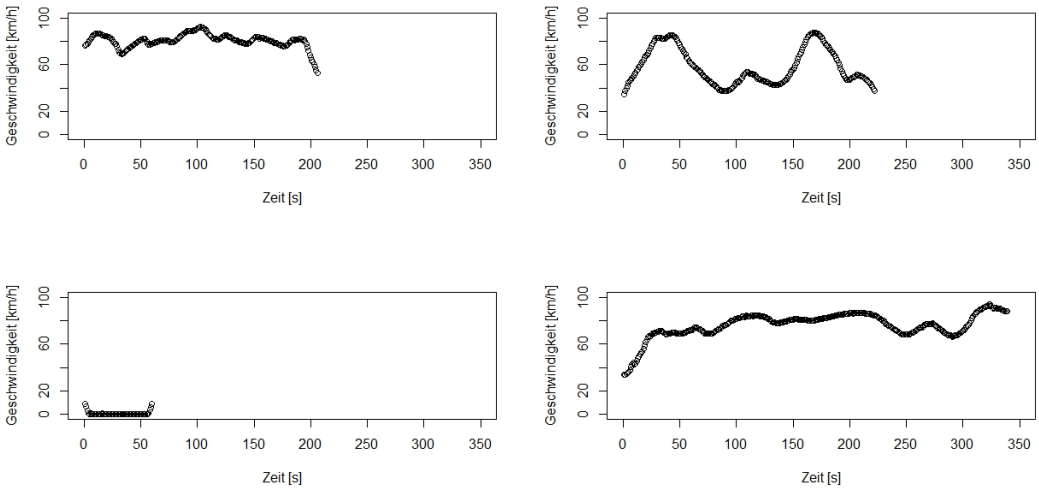


Abbildung A.5.: Passende Parameter für die 1-Norm und damit gefundene Muster (Auswahl)

A.3.3. Distanzmaß 2-Norm

Mit der 2-Norm wurden beinahe die gleichen Muster gefunden wie mit der 1-Norm (der verwendete Seed des Zufallsgenerators war auch derselbe). Auch die Parameter des Algorithmus können mit diesem Distanzmaß wie bei der 1-Norm gewählt werden. Es scheint also keinen Unterschied zu machen ob die 1- oder 2-Norm verwendet wird. Dementsprechend können auch hier die Muster nicht wirklich typischen Fahrsituationen zugeordnet werden.

Parameter	Wert
window size w	500
subwindow size $\bar{w}$	8
k	25
gesampelte Fenster für Kandidatenmenge	10
gesampelte Fenster für Vergleichsmenge / Iteration	10

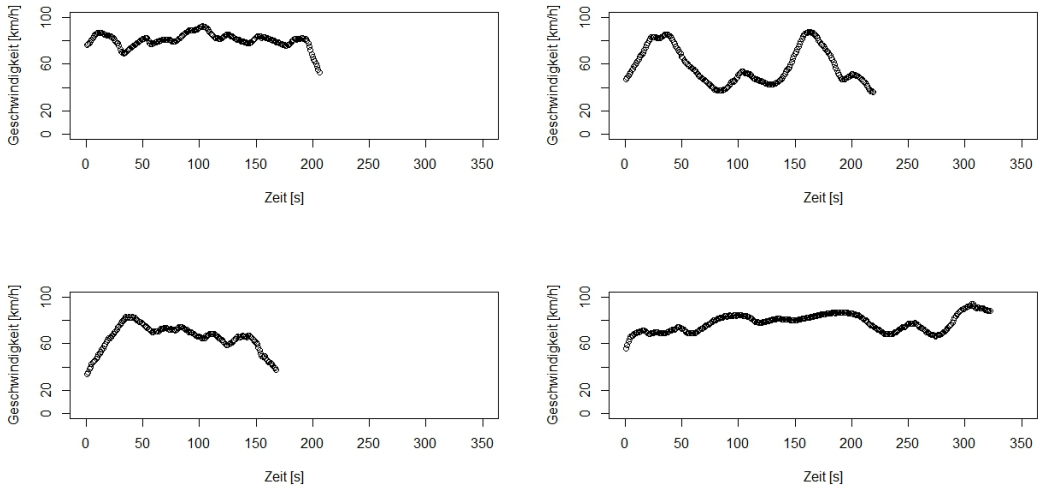


Abbildung A.6.: Passende Parameter für die 2-Norm und damit gefundene Muster (Auswahl)

A.3.4. Distanzmaß Mittelwert + Varianz

Hier wurde versucht die Summe aus Mittelwert und Varianz der punktweisen euklidischen Distanzen von 2 Stücken als Distanzfunktion zu verwenden. Sind sich zwei Stücke ähnlich ist ja sowohl der Mittelwert der Abweichungen als auch die Varianz dieser klein. Sind sie verschieden ist jedoch zumindest eine dieser Komponenten groß.

Unter Verwendung dieses Distanzmaßes werden etwas andere Muster gefunden als mit 1- oder 2-Norm. Die Muster sind im Mittel länger als die mit DTW gefundenen, aber kürzer als die mit 1- und 2-Norm gefundenen. Auch hier sind die entdeckten Muster durchwegs verschieden und lassen keine auf den ersten Blick erkennbaren Gruppierungen zu. Auch die Zuordnung zu typischen Fahrsituationen fällt schwer.

Parameter	**Wert**
window size w	500
subwindow size $\bar{w}$	8
k	25
gesampelte Fenster für Kandidatenmenge	10
gesampelte Fenster für Vergleichsmenge / Iteration	10

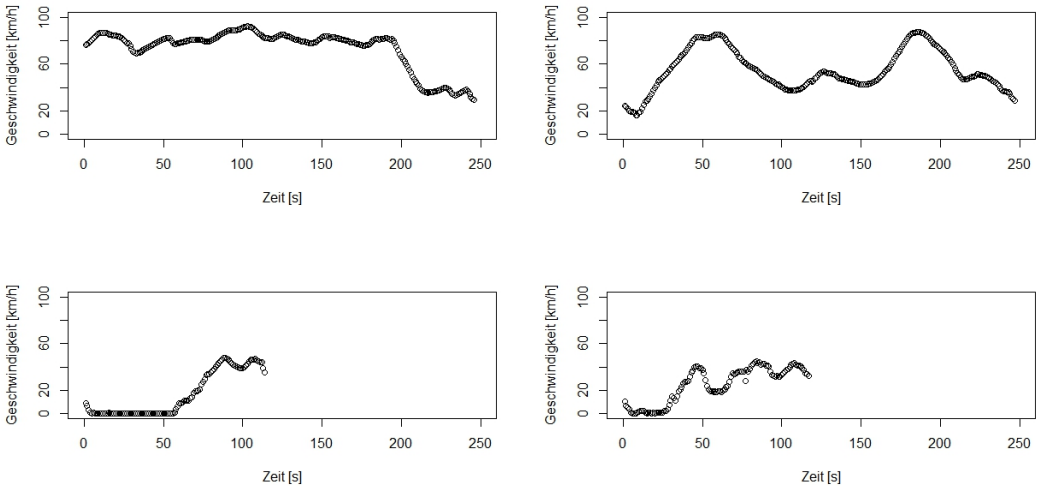


Abbildung A.7.: Passende Parameter für die Verwendung von Mittelwert und Varianz der punktweisen Distanzen als Distanzmaß und damit gefundene Muster (Auswahl)

B. Beispiele von Feature-Vektoren verschiedener Gruppen

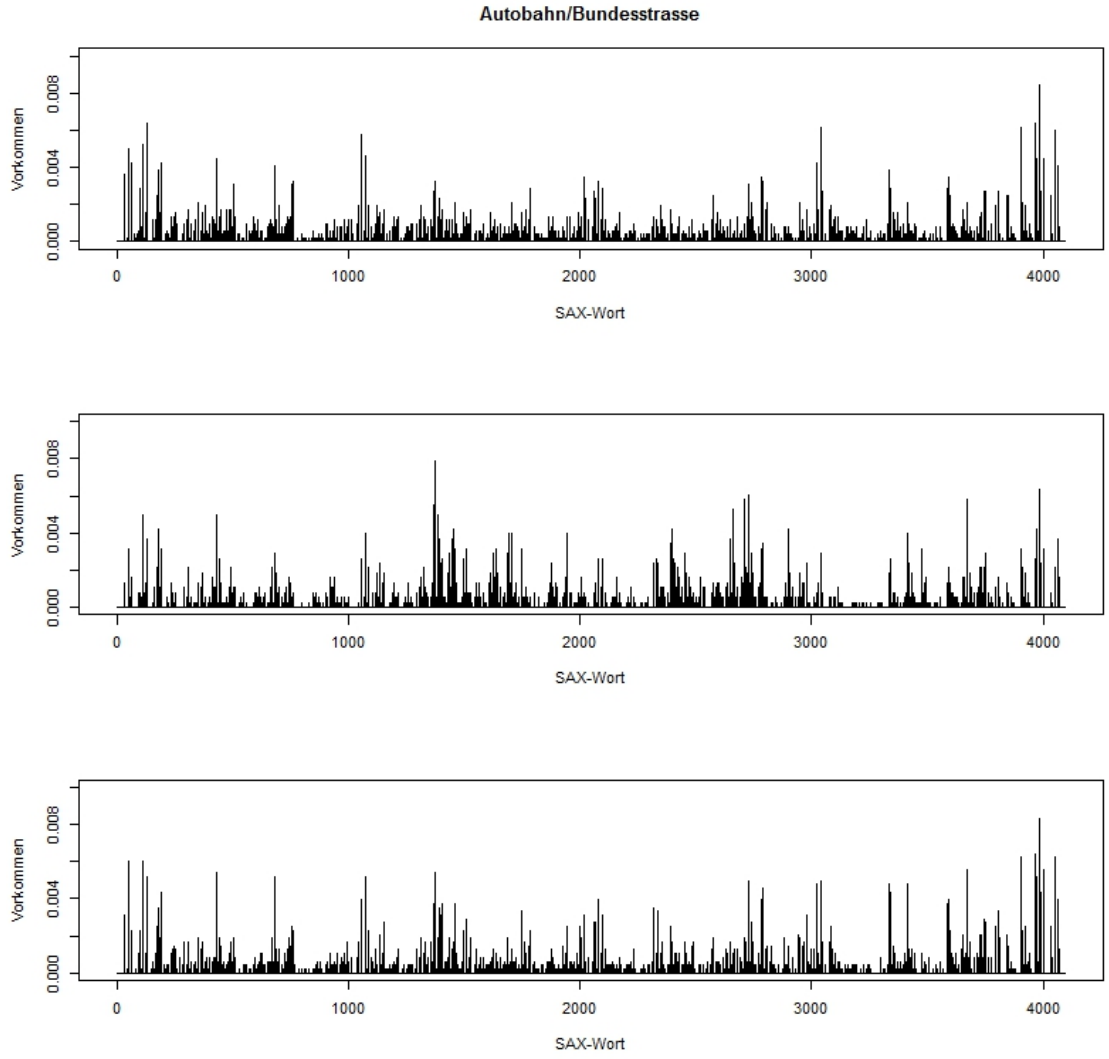


Abbildung B.1.: Feature-Vektoren von Autobahndaten (Auswahl)

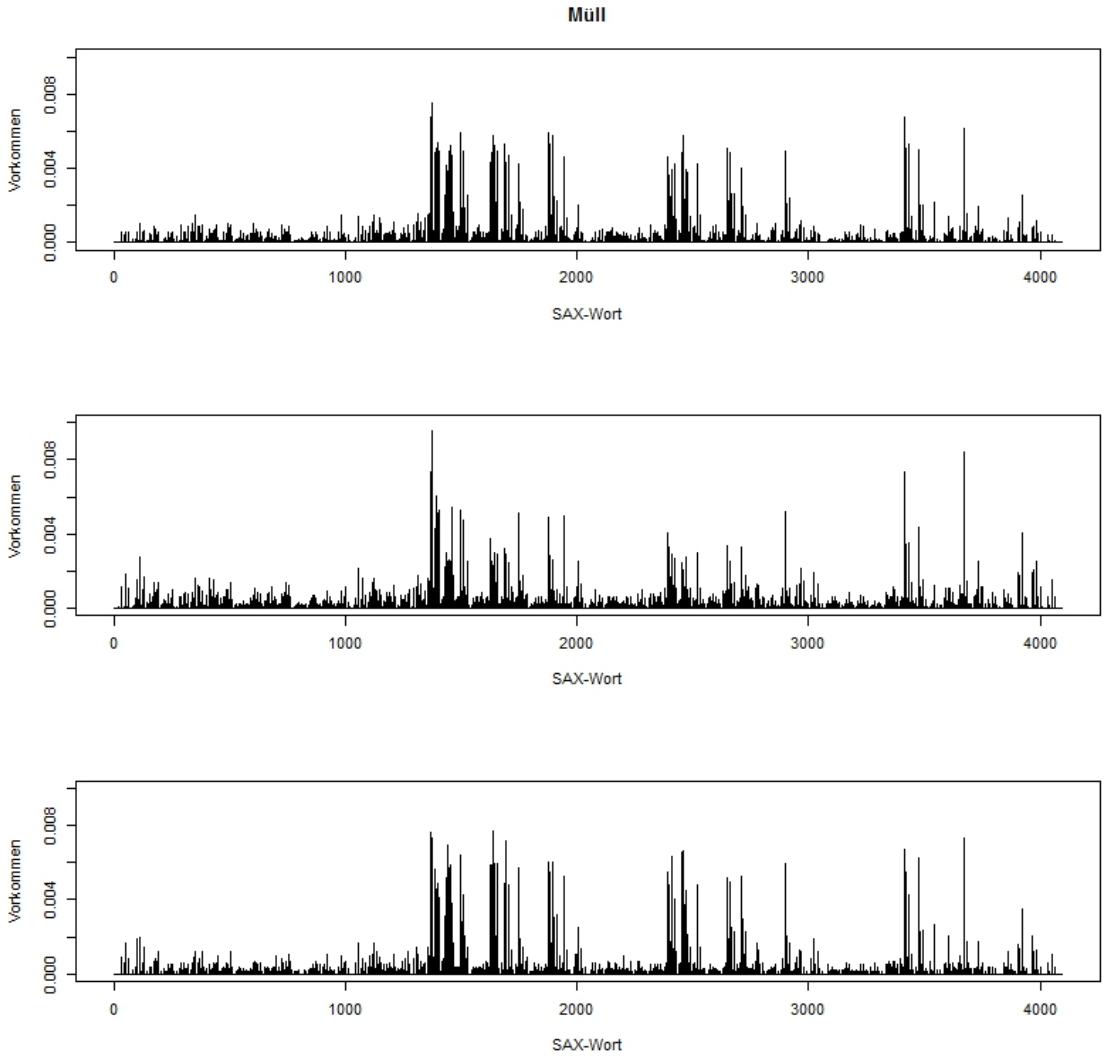


Abbildung B.2.: Feature-Vektoren von Müllfahrzeugdaten (Auswahl)

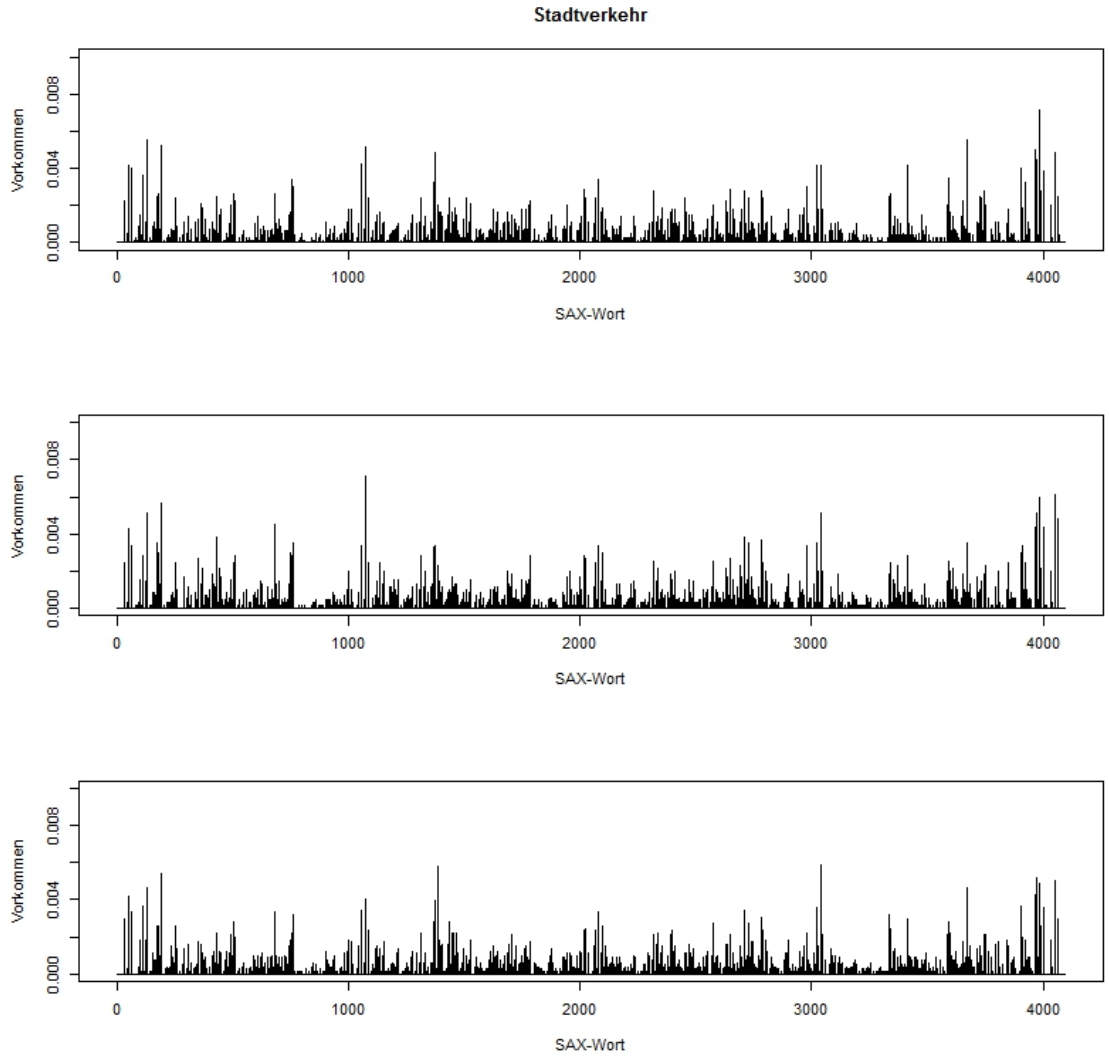


Abbildung B.3.: Feature-Vektoren von Daten von Stadtverkehr (Auswahl)

C. Beispiele der Messungen in ausgewählten Clustern

In diesem Anhang finden sich jeweils 8 zufällig ausgewählte Plots der Fahrzeuggeschwindigkeit für verschiedene Cluster. Die Wahrscheinlichkeit mit der eine Messung gewählt wurde entsprach der Wahrscheinlichkeit, dass sie von der dem Cluster entsprechenden Komponente des GMM erzeugt wurde. Messungen die näher am Clusterzentrum liegen sind also mit größerer Wahrscheinlichkeit zu sehen.

C.1. Müllfahrzeuge

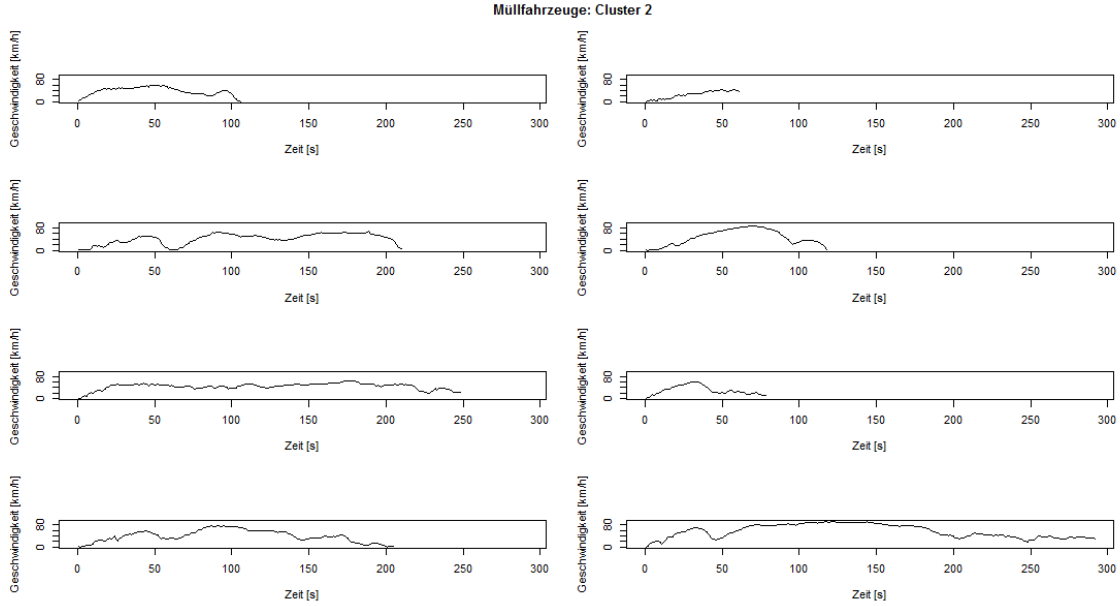


Abbildung C.1.: Cluster 2 der Müllfahrzeug-Daten

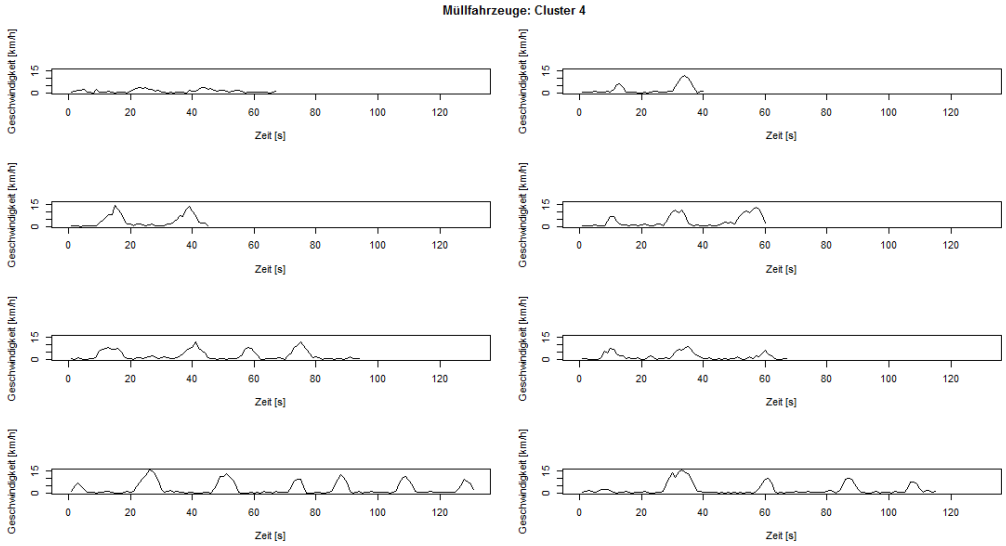


Abbildung C.2.: Cluster 4 der Müllfahrzeug-Daten

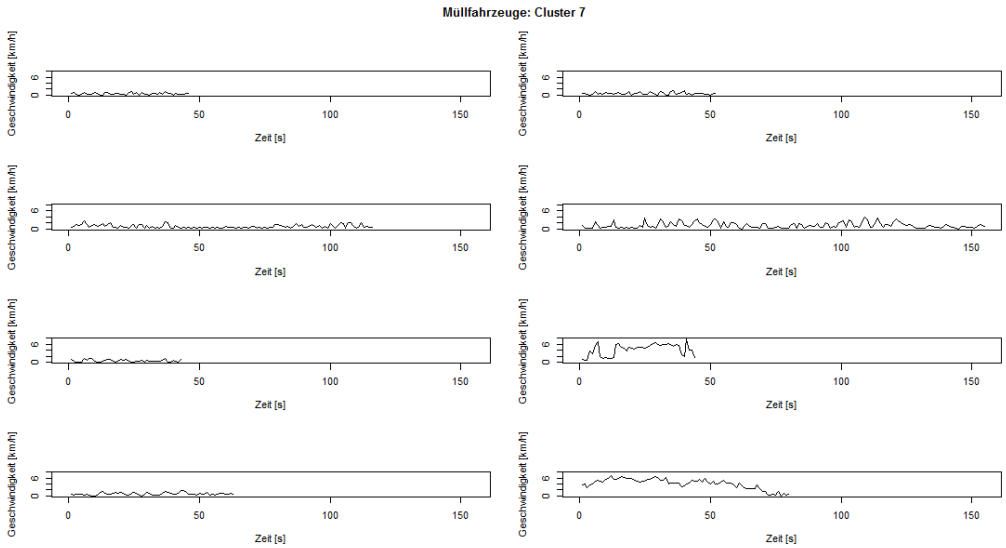


Abbildung C.3.: Cluster 7 der Müllfahrzeug-Daten

C.2. Fahrten auf Autobahn und Bundesstraße

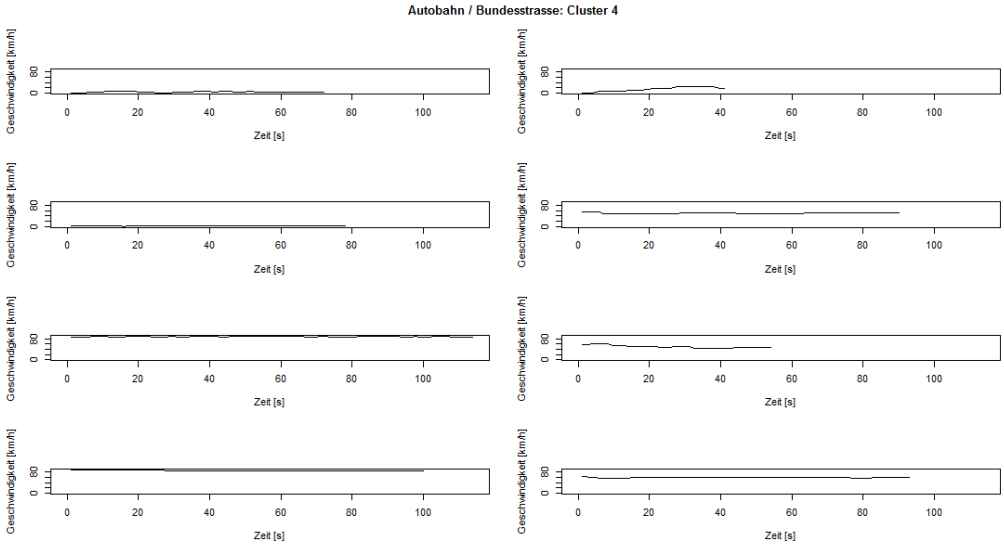


Abbildung C.4.: Cluster 2 der Autobahn/Bundesstraße-Daten

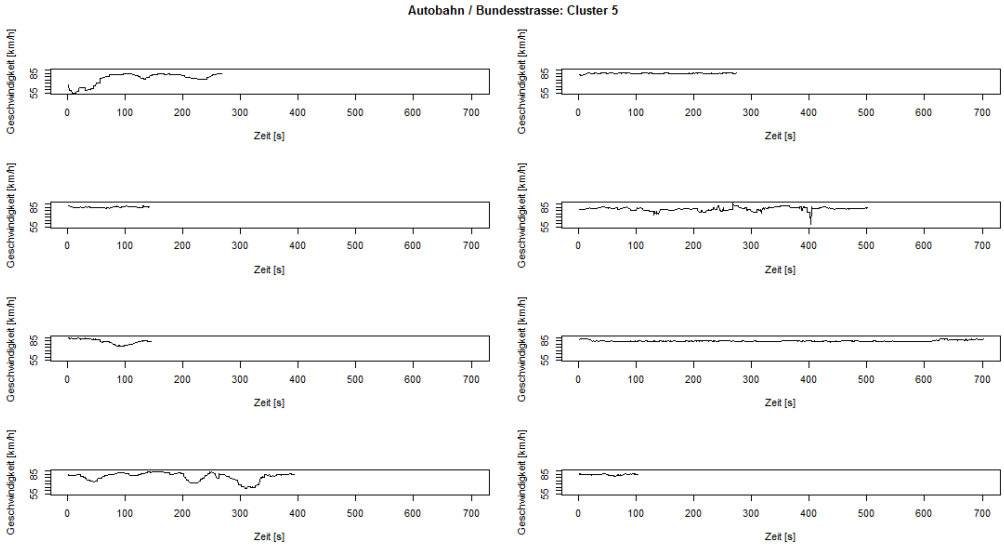


Abbildung C.5.: Cluster 5 der Autobahn/Bundesstraße-Daten

C.3. Stadtverkehr

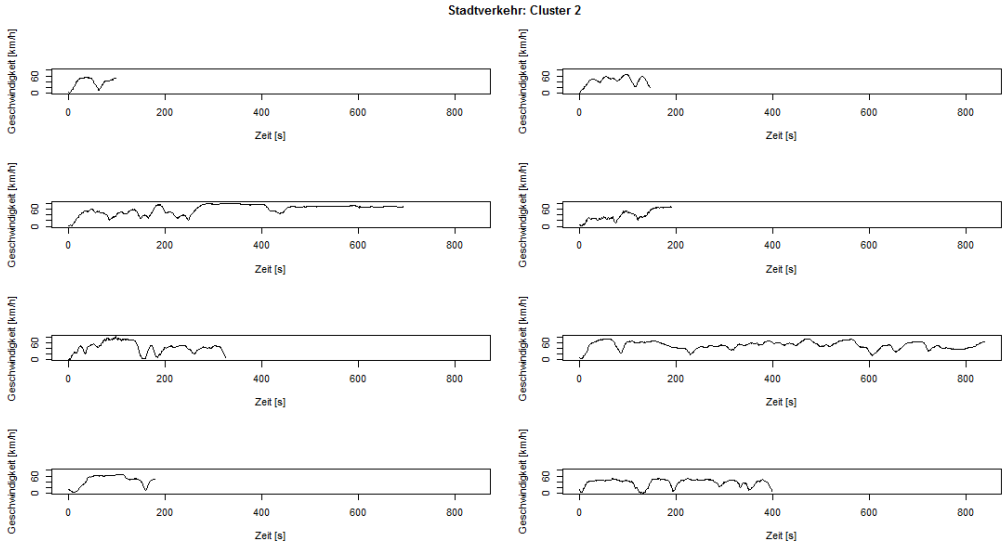


Abbildung C.6.: Cluster 2 der Stadtverkehr-Daten

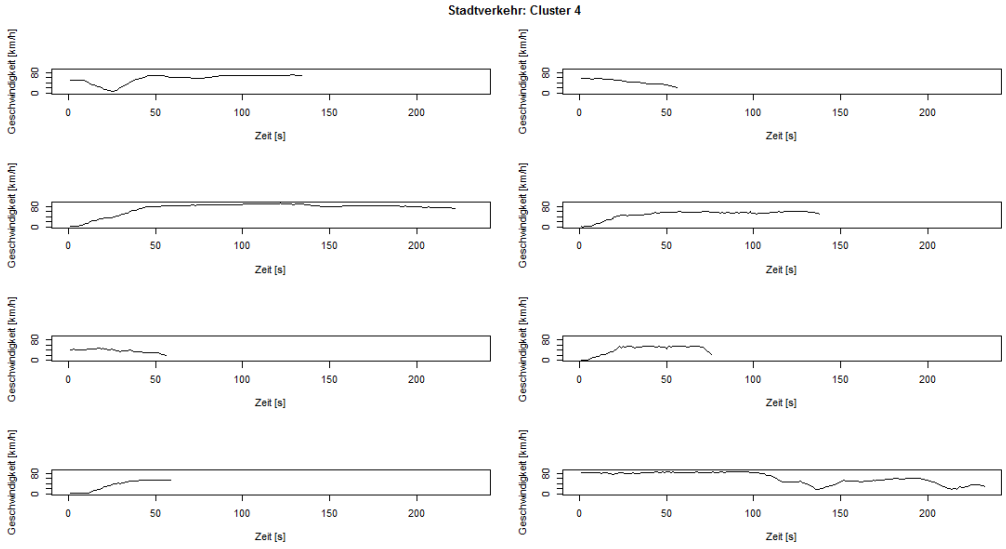


Abbildung C.7.: Cluster 4 der Stadtverkehr-Daten

D. Zur Berechnung verwendeter R-Code

In diesem Anhang sind die im Rahmen der Masterarbeit erstellten Funktionen abgedruckt. Genaue Beschreibungen der Parameter und der Vorgehensweise sind direkt in den Kommentaren im Quellcode

D.1. Code zum Finden von Mustern

Die folgende Funktion implementiert den Algorithmus zur Mustersuche. Zurückgegeben werden Start- und Endindices der gefundenen Muster.

```
##########################################################
# Searches for patterns in data and returns start and
# end indices of found patterns in the data
#
# @param data: numeric vector of data in which to search
# @param wl:   window length of sampled windows
# @param swl:  window length of sliding subwinsows
# @param k:    rate a subwindow by the mean of the k
#              best distances
# @param num_win: number of candidate windows to sample
# @param num_comp_win: number of comparsion windows to
#              sample
# @param max_iter: maximum number of iterations
# @param distance_fun: function that takes two vectors
#              of equal length and returns a single
#              value (the distance between them)
# @param greater_is_better: Do greater values of the
#              distance function indicate greater
#              similarity?
# @return:     A list with two components containing
#              start and end indices of found patterns
#              in the data vector.

findPattern <- function(data, wl, swl, k, num_win, num_comp_win, max_iter, distance
    _fun, greater_is_better = F){

    ##########################################################
    # This helper function samples windows from the data
    # vector, extracts all subwindows and returns them
    # together with their respective start indices.
    getSWSet <- function(data, wl, swl, num_win){
        set <- matrix(ncol=(wl-swl+1)*num_win, nrow=swl);
        start <- rep(0, (wl-swl+1)*num_win);
        ind <- sample.int(length(data)-wl, size=num_win);
        for(i in 1:num_win){
            for(j in 1:(wl-swl+1)){
                set[,(wl-swl+1)*(i-1)+j] <- data[(ind[i]+j-1):((ind[i]+j-1)+swl-1)
                    ];
                #set[,(wl-swl+1)*(i-1)+j] <- (set[,(wl-swl+1)*(i-1)+j]-mean(set[,(
                    wl-swl+1)*(i-1)+j]))/sd(set[,(wl-swl+1)*(i-1)+j])
                start[(wl-swl+1)*(i-1)+j] <- ind[i]+j-1;
            }
        }
        return(list(swMatrix = set, swStart = start));
    }
```

```
43      ########################################################
44      # This helper function takes a subwindow, compares it to
45      # all comparison subwindows and returns the mean of the
46      # k best distances.
47      calcCompDists <- function(sw, compSW){
48          d <- apply(compSW, 2, function(x) distance_fun(x,sw));
49          if(greater_is_better){
50              return(mean(tail(sort(d, partial=((-k+1):0)+length(d)),k)));
51          }
52          return(mean(head(sort(d, partial=1:k),k)));
53      }
54
55      #Sample candidate set
56      candSW <- getSWSet(data, wl, swl, num_win);
57      candSWstart <- candSW$swStart;
58      candSW <- candSW$swMatrix;
59
60      #calculate noise distribution and threshold
61      noise_win <- sample(data, replace = T, size = wl);
62      noiseSW <- matrix(ncol=(wl-swl+1), nrow=swl);
63      for(i in 1:(wl-swl+1)){
64          noiseSW[,i] <- noise_win[i:(i+swl-1)];
65      }
66      print(paste("Calculating noise distribution on", dim(noiseSW)[2], "noise
            subwindows."));
67      flush.console();
68      compSW <- getSWSet(data, wl, swl, num_comp_win)$swMatrix;
69      noise_dist <- sfApply(noiseSW, 2, function(x) calcCompDists(x, compSW));
70      thresh <- ifelse(greater_is_better, quantile(noise_dist, .99), quantile(noise_
            dist, .01));
71      print(paste("Establishing Threshold at", thresh))
72
73      #compare subwindows
74      to_remove <- 0;
75      const_counter <- 0;
76      for(i in 1:max_iter){
77          if(i > 1 & length(to_remove) == 0){
78              const_counter <- const_counter + 1
79          }
80          else{
81              const_counter <- 0;
82          }
83          if(const_counter > 4){
84              break;
85          }
86          print(paste("Iteration",i,": Comparing", length(candSWstart),"candidate
                subwindows to", dim(compSW)[2], "comparison subwindows."));
87          flush.console();
88          compSW <- getSWSet(data, wl, swl, num_comp_win)$swMatrix;
89          distances <- sfApply(candSW, 2, function(x) calcCompDists(x, compSW));
90          if(greater_is_better){
91              to_remove <- which(distances<thresh);
92          }
93          else{
94              to_remove <- which(distances>thresh);
95          }
96          if(length(to_remove)>0){
97              candSWstart <- candSWstart[-to_remove];
98              candSW <- candSW[, -to_remove];
99          }
100         if(length(candSWstart) < 1){
101             break;
102         }
103     }
104
105     #combine overlapping patterns and find endpoints
```

```
106     pattern <- rep(0, length(data));
107     for (ind in candSWstart){
108         pattern[ind:(ind+swl-1)] <- 1;
109     }
110     run_lengths <- rle(pattern);
111     change_inds <- cumsum(run_lengths$lengths);
112     patterns_start <- change_inds[which(run_lengths$values==0)]+1;
113     patterns_end <- change_inds[which(run_lengths$values==1)];
114     patterns_start <- patterns_start[1:length(patterns_end)];
115 
116     return(list(start = patterns_start, end = patterns_end));
117 }
```

Quellcode D.1: Funktion zum Finden der Muster

Der folgende Code zeigt wie die „findPattern“ Funktion zu benutzen ist.

```
1  ##########################################################
2  # Example how to use the findPattern function on
3  # multiple cores. We assume the function has been
4  # sourced, there is a 'data' vector and the 'snowfall'
5  # package has been loaded.
6  
7  #define the parameters:
8  wl <- 100;
9  swl <- 8;
10 k <- 3;
11 num_cand_win <- 10;
12 num_comp_win <- 10;
13 max_iter  <- 25;
14 
15 #define a distance/similarity function
16 distance_function <- function(x,y) dtw(x, y, distance.only=TRUE, keep.internals=
       FALSE, step.pattern=symmetric2)$normalizedDistance
17 
18 #other examples:
19 #distance_function <- cor #be sure to set greater_is_better to TRUE!
20 #distance_function <- function(x,y) sum((x-y)^2)
21 
22 #initialise cluster on 3 cpus
23 sfInit(parallel = TRUE, cpus = 3);
24 
25 #export local environment to the cluster and load the 'dtw' library
26 sfExportAll();
27 sfLibrary(dtw);
28 
29 #calculate result and stop cluster
30 result <- findPattern(data, wl, swl, k, num_cand_win, num_comp_win, max_iter,
       distance_function, greater_is_better=F)
31 sfStop();
32 
33 #Plot the found patterns
34 patterns_start <- result$start
35 patterns_end <- result$end
36 for(i in 1:length(patterns_start))plot(data[patterns_start[i]:patterns_end[i]],
       type="b", main=paste("Muster ", i))
```

Quellcode D.2: Beispielcode zur Benutzung der „findPattern“ Funktion

D.2. Erzeugung von SAX Feature-Vektoren

Die folgenden drei Funktionen erzeugen die verschiedenen Varianten der SAX Feature-Vektoren:

```r
#########################################################
# This function computes the "bag-of-patterns"
# representation for a vector from normalised windows.
#
# @param data: The vector to represent
# @param wl: The length of the sliding window
# @param w: The number of PAA segments / the word length
# @param alpha: The number of symbols to use
# @return: A vector representing the bag-of-patterns
#          histogram.

createFeatureVector <- function(data, wl, w, alpha){

    #set breakpoints and create an empty vector
    breakpoints <- qnorm(seq(1/alpha,1,1/alpha))[1:(alpha-1)]
    vec <- rep(0,alpha^w)
    stopifnot(wl%%w==0)

    #stores last sax word (consecutive occurences of the same
    #word are only counted once
    prevsaxind <- -1

    powers_of_alpha <- alpha^(0:(w-1))

    for(i in 1:(length(data)-wl+1)){
        #extract and normalize window
        sw <- data[i:(i+wl-1)]
        sw <- sw-mean(sw)
        swsd <- sd(sw)
        if(swsd > 0) sw <- sw/swsd
        #calculate PAA representation
        paa <- rowMeans(matrix(sw, nrow=w, byrow=T))
        #find the intervals in which the PAA values fall and compute the resulting
            index (plus 1
        #because R indexing starts with 1)
        saxind <- sum(.Internal(findInterval(breakpoints, paa, FALSE, FALSE))*
            powers_of_alpha) + 1
        #increase component by one if we found a "new" word
        if(saxind != prevsaxind){
            vec[saxind] <- vec[saxind] + 1
            prevsaxind <- saxind
        }
    }
    return(vec/sum(vec))
}
```

Quellcode D.3: Funktion zum Erzeugen der SAX Feature-Vektoren (normalisierte Fenster)

```r
#########################################################
# This function computes the "bag-of-patterns"
# representation for a vector from unnormalised windows
# by using custom breakpoints.
#
# @param data: The vector to represent
# @param wl: The length of the sliding window
# @param w: The number of PAA segments / the word length
# @param breakpoints: The breakpoints to use
# @return: A vector representing the bag-of-patterns
#          histogram.

createFeatureVectorBP <- function(data, wl, w, breakpoints){

    #for using the .Internal version of findInterval we
    #must ensure that the breakpoints are sorted
    breakpoints <- sorted(unique(breakpoints))
```

```
18
19      #set alpha and create an empty vector
20      alpha <- length(breakpoints) + 1
21      vec <- rep(0,alpha^w)
22      stopifnot(wl%%w==0)
23
24      #stores last sax word (consecutive occurences of the same
25      #word are only counted once
26      prevsaxind <- -1
27
28      powers_of_alpha <- alpha^(0:(w-1))
29
30      for(i in 1:(length(data)-wl+1)){
31          #extract window
32          sw <- data[i:(i+wl-1)]
33          #calculate PAA representation
34          paa <- rowMeans(matrix(sw, nrow=w, byrow=T))
35          #find the intervals in which the PAA values fall and compute the resulting
                index (plus 1
36          #because R indexing starts with 1)
37          saxind <- sum(.Internal(findInterval(breakpoints, paa, FALSE, FALSE))*
                powers_of_alpha) + 1
38          #increase component by one if we found a "new" word
39          if(saxind != prevsaxind){
40              vec[saxind] <- vec[saxind] + 1
41              prevsaxind <- saxind
42          }
43      }
44      #for tf-idf we can use
45      #return(.5+.5*vec/max(vec)) #instead.
46      return(vec/sum(vec))
47  }
```

Quellcode D.4: Funktion zum Erzeugen der SAX Feature-Vektoren (nicht normalisierte Fenster)

```
1   ##########################################################
2   # This function computes the "bag-of-patterns"
3   # representation for 3 vectors from unnormalised windows
4   # by using custom breakpoints and a 3D version of SAX
5   #
6   # @param data1/2/3: The vectors to represent
7   # @param bp1/2/3: Custom breakpoints for ech vector
8   # @param wl: The length of the sliding window
9   # @param w: The number of PAA segments / the word length
10  # @return: A vector representing the bag-of-patterns
11  #          histogram.
12
13  createFeatureVector3D <- function(data1, bp1, data2, bp2, data3, bp3, wl, w){
14
15      #ensure breakpoints are unique and sorted (for .Internal version
16      #of findInterval
17      bp1 <- sorted(unique(bp1))
18      bp2 <- sorted(unique(bp2))
19      bp3 <- sorted(unique(bp3))
20
21      #ensure the data vectors and breakpoints have the same length
22      stopifnot(length(bp1)==length(bp2) & length(bp2)==length(bp3))
23      stopifnot(length(data1)==length(data2) & length(data2)==length(data3))
24      stopifnot(wl%%w==0)
25
26      alpha <- length(bp1) + 1
27      vec <- rep(0,alpha^(3*w))
28      prevsaxind <- -1
29
30
```

```
    powers_of_alpha1 <- alpha^(0:(w-1))
    powers_of_alpha2 <- alpha^(w:(2*w-1))
    powers_of_alpha3 <- alpha^((2*w):(3*w-1))

    for(i in 1:(length(data1)-wl+1)){
        #extract windows
        sw1 <- data1[i:(i+wl-1)]
        sw2 <- data2[i:(i+wl-1)]
        sw3 <- data3[i:(i+wl-1)]

        #calculate PAA representations
        paa1 <- rowMeans(matrix(sw1, nrow=w, byrow=T))
        paa2 <- rowMeans(matrix(sw2, nrow=w, byrow=T))
        paa3 <- rowMeans(matrix(sw3, nrow=w, byrow=T))

        #find the intervals in which the PAA values fall and compute the resulting
            index
        saxind1 <- sum(.Internal(findInterval(bp1, paa1, FALSE, FALSE))*powers_of_
            alpha1)
        saxind2 <- sum(.Internal(findInterval(bp2, paa2, FALSE, FALSE))*powers_of_
            alpha2)
        saxind3 <- sum(.Internal(findInterval(bp3, paa3, FALSE, FALSE))*powers_of_
            alpha3)

        #R vectors start with 1
        saxind <- saxind1 + saxind2 + saxind3 + 1

        #increase component by one if we found a "new" word
        if(saxind != prevsaxind){
            vec[saxind] <- vec[saxind] + 1
            prevsaxind <- saxind
        }
    }
    return(.5+.5*vec/max(vec)) # tf-idf variant
}
```

Quellcode D.5: Funktion zum Erzeugen der 3D-SAX Feature-Vektoren

Die nächste Funktion kann zum Vorbereiten der Daten für die Erzeugung der Feature-Vektoren verwendet werden.

```
#########################################################
# This function takes a data frame (a measurement)
# prepares it and computes its feature-vector (different
# variants). We assume the SAX-Parameters wl, w, alpha
# have already been set and createFeatureVector
# have been sourced. Only use one of the listed variants
# at a time!
#
# @param x: data frame that represents a measurement and
#           contains the columns "vehicle_speed",
#           "engine_speed" and "engine_torque"

prepareFV <- function(x){

    #find complete cases and ensure there are a minimum of 300
    c <- complete.cases(x$vehicle_speed, x$engine_speed, x$engine_torque)
    if(sum(c) < 300) return(NULL)

    ######################################################
    # normalised windows / standard SAX for vehicle speed
    return(createFeatureVector(x[c, "vehicle_speed"], wl, w, alpha))

    ######################################################
    # unnormalised windows / custom breakpoints with
    # different channels alone
```

```
26      return(createFeatureVector(x[c, "vehicle_speed"], wl, w, alpha, c(25, 50, 75)))
27      return(createFeatureVector(x[c, "engine_speed"], wl, w, alpha, c(900, 1300,
            1800)))
28      return(createFeatureVector(x[c, "engine_torque"], wl, w, alpha, c(-1500, 0,
            1500)))
29
30      ######################################################
31      # 3 channels combined with PCA
32      x <- scale(x[c,c("vehicle_speed", "engine_speed", "engine_torque")])
33      e <- eigen(cov(x), symmetric = T)
34      me <- max(e$values)
35      v <- e$vectors[, which(e$values==me)]
36      x <- x%*%v
37      return(createFeatureVector(x, wl, w, alpha))
38
39      ######################################################
40      # 3 channels combined by concatenating the
41      # feature-vectors, custom breakpoints
42      res <- createFeatureVector(x[c, "vehicle_speed"], wl, w, alpha, c(25, 50, 75))
43      res <- c(res, createFeatureVector(x[c, "engine_speed"], wl, w, alpha, c(900,
            1300, 1800)))
44      res <- c(res, createFeatureVector(x[c, "engine_torque"], wl, w, alpha, c(-1500,
             0, 1500)))
45      return(res)
46
47      ######################################################
48      # 3 channels combined by using 3D SAX
49      return(createFeatureVector3D(x[c, "vehicle_speed"], c(25,50,75),
50                                   x[c, "engine_torque"], c(-1500,0,1500),
51                                   x[c, "engine_speed"], c(900,1300,1800), wl, w))
52  }
```

Quellcode D.6: Funktion zum Vorbereiten der Daten für die Erzeugung der Feature-Vektoren

Mit diesem Code kann aus einer Liste von als „Data-Frame“ vorhandenen Messungen die zugehörige Matrix der Feature-Vektoren erzeugt werden. Jede Zeile der Matrix korrespondiert zu einer Messung, jede Spalte zu einem SAX-Wort. Auch die mit tf-idf gewichtete Variante der Feature-Vektoren kann mit diesem Code erzeugt werden.

```
1  ######################################################
2  # Example how to create feature-vectors for a list of
3  # measurements (data frames) on mutiple cores. We assume
4  # package 'snowfall' has been loaded, needed functions
5  # (esp. 'prepareFV') have been sourced and the
6  # SAX-Parameters have been set.
7
8  #Initialise cluster on 4 cores and export data to workers
9  sfInit(parallel = TRUE, cpus = 4)
10 sfExportAll()
11
12 #MuellDFs is a list of data frames containing measurements
13
14 #call 'prepareFV' for each data frame
15 MuellFV <- lapply(MuellDFs,prepareFV)
16 #remove NULL values (measurements with less than 300 valid points)
17 MuellFV[sapply(MuellFV, is.null)] <- NULL
18 #make a matrix whose rows correspond to the measurements and whose
19 #columns are the bag-of-patterns bins.
20 MuellFV <- do.call(rbind, MuellFV
21
22 #tf-idf weights if wanted:
23
24 #combine all matrices of feature-vectors to one big matrix:
25 fvmat <- rbind(MuellFV, ...)
```

```
26
27 idf <- log(nrow(fvmat)/(1+colSums(fvmat!=.5))) #when using standard relative
       frequencies replace .5 with 0
28 fvmat <- t(t(fvmat) * idf)
```

Quellcode D.7: Code zur Erzeugung der Feature-Vektoren unter Verwendung der vorher definierten Funktionen.

D.3. Splitten der Messungen in Teile

Die folgende Funktion wird zum Splitten einer einzelnen Messung verwendet. Es muss bereits ein bool'scher Vektor gegeben sein, der punktweise bestimmt ob an dieser Stelle getrennt werden darf. Die Funktion bestimmt Teile die direkt eine Mindestanzahl von aufeinanderfolgenden WAHR-Werten besitzen und durch eine Mindestanzahl FALSCH-Werte getrennt sind.

```
1  ##########################################################
2  # Given a boolean vector, the function finds segments of
3  # true-values that have a given minimum length and are
4  # separated by segments of false-values that also must
5  # have a minimum length.
6  #
7  # @param to_use: boolean vector
8  # @param min_length_good: minimum length of segments
9  #          with true-values
10 # @param min_length_bad: minimum length of segments with
11 #          false-values
12 # @return: a factor that indicates segments of
13 #          true-values that fulfill the conditions. Each
14 #          segment gets its own factor-level. Level 0
15 #          indicates points that do not belong to a
16 #          segment.
17
18
19 getSplitFactor <- function(to_use, min_length_good=40, min_length_bad=5){
20     stopifnot(is.logical(to_use))
21     enc <- rle(as.numeric(to_use))
22
23     #make false-values which are in too short segments true
24     enc$values[enc$lengths < min_length_bad] <- 1
25     enc <- rle(inverse.rle(enc))
26
27     #make true-values which are in too short segments false
28     enc$values[enc$lengths < min_length_good] <- 0
29     good_inds <- enc$values==1
30
31     #Compute factor levels and return factor
32     enc$values[good_inds] <- 1:sum(good_inds)
33     return(as.factor(inverse.rle(enc)))
34 }
```

Quellcode D.8: Funktion zum Splitten einer einzigen Messung.

Mit den folgenden Funktionen kann nun eine ganze Liste von data-frames von Messungen in ihre Segmente zerteilt werden:

```
1  ##########################################################
2  # This function takes a data frame containing a
3  # measurement and splits it to segments where either
4  # vehicle speed, engine speed or engine torque fall below
5  # a given threshold. We assume the function
```

```r
# 'getSplitFactor' has been sourced
#
# @param DF: the data frame containg the measurement
# @param vspeed: Threshold for vehicle speed
# @param espeed: Threshold for engine speed
# @param torque: Threshold for engine torque

splitDF <- function(DF, vspeed=.5, espeed=100, torque=50){
    c <- complete.cases(DF$vehicle_speed, DF$engine_speed, DF$engine_torque)
    DF <- DF[c, c("vehicle_speed", "engine_speed", "engine_torque")]
    sf <- getSplitFactor(DF$vehicle_speed > vspeed &
                         DF$engine_speed > espeed &
                         abs(DF$engine_torque) > torque)
    res <- split(DF, sf)
    res$'0' <- NULL
    return(res)
}

#########################################################
# Just calls 'splitDF' (see above) for a list of data
# frames and concatenates teh result.

splitsFromGroup <- function(DFs){
    do.call(c, lapply(DFs, splitDF))
}

#Example on how to use the functions
MuellSplits <- splitsFromGroup(MuellDFs)
```

Quellcode D.9: Code zum Splitten einer Liste von Messungen.

D.4. Erzeugen der Feature-Vektoren der Teile

Mit der folgenden Funktion können die Feature-Vektoren für die einzelnen Teile bestimmt werden. Die Breakpoints werden als Quartile berechnet.

```r
#########################################################
# This function creates the feature vectors for a group.
# We assume the function 'createFeatureVectorBP' has
# sourced and SAX-Parameters have been set. Breakpoints
# are computed as the quartiles of the data.
#
# @param Splits: List of segments to create the feature
#                vectors for. For example createt by the
#                function 'splitsFromGroup'.
# @return: A matrix where the i-th row is the feature vector
#          of the i-th segment.

prepareSplitFVs <- function(Splits){

    #Concatenate the respective channels for quantile comutation
    vspeeds <- do.call(c, lapply(Splits, function(x) x$vehicle_speed))
    espeeds <- do.call(c, lapply(Splits, function(x) x$engine_speed))
    torques <- do.call(c, lapply(Splits, function(x) x$engine_torque))

    #Compute breakpoints as quantiles
    bp_vspeed <- quantile(vspeeds, c(.25,.5,.75))
    bp_espeed <- quantile(espeeds, c(.25,.5,.75))
    bp_torque <- quantile(torques, c(.25,.5,.75))

    #Create feature vectors for each channel
    vspeedFV <- lapply(Splits, function(x) createFeatureVectorBP(x[, "vehicle_speed
        "], wl, w, bp_vspeed))
```

```
27      vspeedFV <- do.call(rbind, vspeedFV)
28      espeedFV <- lapply(Splits, function(x) createFeatureVectorBP(x[, "engine_speed"
            ], wl, w, bp_espeed))
29      espeedFV <- do.call(rbind, espeedFV)
30      torqueFV <- lapply(Splits, function(x) createFeatureVectorBP(x[, "engine_torque
            "], wl, w, bp_torque))
31      torqueFV <- do.call(rbind, torqueFV)
32
33      #concatenate and return the feature vectors
34      return(cbind(vspeedFV, espeedFV, torqueFV))
35  }
36
37
38  #Example on how to use the function
39  MuellSplitFV <- prepareSplitFVs(MuellSplits)
```

Quellcode D.10: Code zur Erzeugung der Feature-Vektoren von Teilen von Messungen.