HAW HAMBURG

BACHELORTHESIS
Anita Rados

Analyse und Visualisierung von Live-Leistungsdaten im Profisport

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Anita Rados

Analyse und Visualisierung von Live-Leistungsdaten im Profisport

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Informatik Technischer Systeme*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Marina Tropmann-Frick
Zweitgutachter: Prof. Dr. Martin Schultz

Eingereicht am: 02. Juni 2021

Anita Rados

Thema der Arbeit

Analyse und Visualisierung von Live-Leistungsdaten im Profisport- Vergleich von Analysemethoden und prototypische Umsetzung am Beispiel des Profisports

Stichworte

Sport, Fußball, Live-Daten, Clustering, Klassifikation, Visualisierung

Kurzzusammenfassung

Das Ziel der vorliegenden Bachelorarbeit war es, verschiedene Clustering-Verfahren zu vergleichen und basierend auf diesen Klassifikatoren aufzubauen. Für das Clustering wurden Daten aus zwei verschiedenen Datenquellen verwendet. Es wurden insgesamt sieben verschiedene Clustering-Verfahren verwendet und drei unterschiedliche Klassifikatoren. Die Ergebnisse der Klassifikation sollten dann visuell dargestellt werden. Dafür wurden vier verschiedene GUIs gebaut, deren Implementierung sich auf jener der Clusterung und der Klassifikatoren stützt.

Anita Rados

Title of Thesis

Analysis and visualization of live performance data in professional sports- Comparison of analysis methods and prototypical implementation using the example of professional sports

Keywords

Sport, Soccer, Live-Data, Clustering, Classification, Visualisation

Abstract

This Bachelors thesis’s main goal was to compare a variety of clustering algorithms and develop different Classifiers based on those algorithms. Data from two different data sources has been used for the clustering. There has been a total of seven different clustering algorithms used as well as three different classifiers. The results of the classifiers will be visualized in four

different GUIs. The Implementation of these GUIs is based on both the implementations of the clustering algorithms and the classifiers.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Im modernen Fußball stehen den Mannschaften eine Reihe von Daten zu Verfügung, die genutzt werden können, um sich einen spielerischen Vorteil zu verschaffen. Ein Teil dieser Daten besteht aus Statistiken der eigenen oder gegnerischer Mannschaften, sowie Statistiken von einzelnen Spielern. Ein weiterer Teil dieser Daten sind Live-Daten, die während eines laufenden Spiels gesammelt werden. Bei den Live-Daten ist es für den Menschen, im Vergleich zu Statistiken, besonders schwer die in ihnen enthaltenen nützlichen Informationen herauszufiltern. Damit die Daten nicht nur aufgezeichnet werden, sondern die nützlichen Informationen auch aus ihnen gewonnen werden können, müssen sie in einen Kontext gebracht werden. Eine Möglichkeit dies zu tun ist es die Live-Daten in Relation zu schon vorhandenen, vorher gesammelten, Daten zu setzen und diese Informationen während des laufenden Spiels anschaulich an die entsprechenden Entscheidungsträger zu leiten. Um die Erkenntnisse möglichst verständlich an die Entscheidungsträger zu leiten, ist beispielsweise eine einfache GUI notwendig. Dadurch wird es diesen ermöglicht Abweichungen oder Auffälligkeiten der in den Live-Daten enthaltenen Event- und Positionsdaten zu erkennen. Mit Hilfe dieser neu gewonnenen Informationen kann dann versucht werden Einfluss auf das Spielgeschehen zu nehmen. Im Rahmen dieser Arbeit werden dabei Live-Daten der HSV Fußball AG aufgezeichnet, diese werden im Kontext zu vorher aufgezeichnete Live-Daten gebracht, damit mit Hilfe verschiedener Methoden Abweichungen deutlich gemacht werden können. Hierbei ist zu beachten, dass bisher keine Aufzeichnung der Daten erfolgte.

Die Arbeit soll es für die Zukunft möglich machen, auf den Ergebnissen der Arbeit aufzubauen und möglicherweise ein ausgereiftes System zu erschaffen, welches dann im Spielbetrieb genutzt werden kann.

Die Arbeit soll Live-Daten der Spieler, wie die durchschnittliche Geschwindigkeit, mit schon vorhandenen Daten vergleichen und den aktuellen Wert einem Cluster und damit einer Kategorie zuordnen. Hierdurch sollen die zuständigen Mitarbeiter, während eines Live-Spiels über die aktuelle Leistung der Spieler genauer informiert werden.

Es sollen verschiedene Methoden der Clusterung angewendet werden, sowohl Clustering-Verfahren mit vorgegebener Anzahl an Clustern als auch Clustering-Verfahren, welche die Anzahl der Cluster selbst bestimmen. Auch sollen verschiedenen Klassifikatoren für die Zuordnung der Werte benutzt werden.

1.2 Zentrale Fragestellung:

Dabei gibt es zwei zentrale Fragestellungen, sie lauten wie folgt:

1. Welche statistischen Methoden, sowie Methoden des Data Mining und des maschinellen Lernens sind zu implementieren, um den Bedarfen gerecht zu werden.

Hierbei soll herausgefunden werden, welche Methoden unter welchen Bedingungen geeignet sind und welche davon das beste Ergebnis liefert, welche Methoden unter welchen Bedingungen nicht geeignet sind und welche Methoden unter keinen Umständen geeignet sind.

Für die Darstellung der Ergebnisse ergibt sich folgende Fragestellung:

2. Wie können diese Ergebnisse dargestellt und übermittelt werden, um den Anforderungen zu genügen.

Hierbei soll ermittelt werden, welche Darstellungsarten für die verschiedenen Ergebnisse am besten geeignet ist.

1.3 Aufbau der Ausarbeitung

Das Kapitel 2 *Grundlagen und Exploration der dfldata* befasst sich hauptsächlich mit dem Clustering der Daten aus der Datenbank *dfldata*. Am Beginn werden, für das Verständnis, die sportspezifischen Regeln und Begriffe geklärt. Darauf folgt eine Beschreibung der Datenbank *dfldata* sowie der Attribute, mit welchen sich diese Arbeit unter anderem befassen wird. Anschließend wird die Datenbeschaffung aus der *dfldata* und die dazu nötigen Implementierungen beschrieben. Um mit den Daten der *dfldata* vertraut zu werden und mögliche Besonderheiten dieser aufzudecken folgt daraufhin die explorative Analyse der Daten. Mit den gewonnenen Erkenntnissen werden im nachfolgenden Abschnitt Überlegungen über die mögliche zweite

Dimension der Daten in Bezug auf die Clusterung gemacht. Der Kern des Kapitels befindet sich in dem Abschnitt *Clustering*, hier werden Verschiedene Clustering-Verfahren angewendet und die Ergebnisse dieser festgehalten, um das optimale Verfahren für den geplanten Klassifikator finden. Die Entscheidung, welche oder welches Verfahren gewählt wird erfolgt in dem Abschnitt *Diskussion*.

Das Kapitel 3 *Exploration der DFL-Daten* beschäftigt sich wie das vorherige Kapitel primär mit dem Clustering, allerdings wird hier mit einer anderen Datenquelle gearbeitet. Es erfolgt zunächst die Beschreibung der URLs, welche die Datenquellen bilden. Der darauffolgende Abschnitt beschäftigt sich mit der Datenbeschaffung und den dazu nötigen Implementierungen. Daraufhin werden in dem Abschnitt *Clustering* dieselben Clustering-Verfahren wie aus Kapitel 2 angewendet und ihre Ergebnisse festgehalten. Die Entscheidung, welche oder welches Verfahren für die Klassifikatoren gewählt wird erfolgt in dem letzten Abschnitt *Diskussion*.

Im Kapitel 4 *Klassifikator* werden die in Klassifikatoren beschrieben, welche unter anderem auf den Ergebnissen aus Kapitel 2 und 3 aufbauen. Dafür werden zunächst die Implementierungen geklärt. Eine dieser Implementierungen bezieht sich nicht auf die Ergebnisse aus den Kapiteln 2 und 3 sondern basiert auf der Standardabweichung. Darauf folgt eine Auswertung der Klassifikatoren, welche auf den Kapiteln 2 und 3 basieren in dem Abschnitt *Ergebnisse* und eine Bewertung der erzielten Ergebnisse in dem Abschnitt *Diskussion*.

Das Kapitel 5 *Visualisierung* beschäftigt sich mit der visuellen Darstellung der aus der Klassifikation erzielten Ergebnisse. Dieses Kapitel ist in die Abschnitte *GUI Standardabweichung* und *GUI Klassifikator* unterteilt. Hierbei wird in dem Abschnitt *GUI Standardabweichung* auf GUIs eingegangen, welche auf der in Kapitel 4 erwähnten Implementierung auf Basis der Standardabweichung beruhen. Dafür werden zunächst die GUIs und ihre Funktionalitäten vorgestellt und anschließend wird auf die Implementierung dieser eingegangen. Im Abschnitt *GUI Klassifikator* wird auf die GUIs, welche auf Basis der Ergebnisse der Clusterung und der Klassifikation beruhen, eingegangen. Es werden ebenfalls zunächst die GUIs und ihre Funktionalitäten vorgestellt und im Anschluss wird auf die Implementierungen eingegangen.

Im Kapitel 6 *Fazit* werden die erzielten Ergebnisse kurz bewertet und ein Ausblick gegeben.

2 Grundlagen und Exploration der *dfldata*

2.1 Theoretische Grundlagen zum Sport

In dieser Arbeit werden Spiele der 2. Deutschen Bundesliga betrachtet. Die Spiele werden hierbei in Saisons unterteilt, wobei eine Saison regulär im Juli eines Jahres startet und im Mai des Folgejahres endet. Diese Arbeit beschäftigt sich mit den Spielen der Saison 2019/2020 sowie 2020/2021. Die Saison 2019/2020 startete im Juli 2019 und endete coronabedingt im Juni 2020. Auch auf die Saison 2020/2021 hatte die Coronapandemie Auswirkungen, diese begann im September 2020 und soll im Mai 2020 enden. Eine Saison beinhaltet dabei 34 Spieltage, wobei an jedem Spieltag neun Spiele stattfinden. Jede Mannschaft der Liga absolviert dementsprechend 34 Spiele. Ein Spieltag kann mehrere Daten erfassen, muss also nicht an einem einzigen Datum stattfinden. In der 2. Deutschen Bundesliga beinhaltet ein Fußballspiel 90 Minuten reguläre Spielzeit, diese ist in zwei Hälften von jeweils 45 Minuten regulärer Spielzeit unterteilt. Zusätzlich zur regulären Spielzeit kann es am Ende der jeweiligen Hälften noch Nachspielzeit geben, welche aus einer oder mehreren Minuten bestehen kann. Zwischen den zwei Halbzeiten ist eine Pause von 15 Minuten vorgesehen. In Spielen der 2. Deutschen Bundesliga sind seit der Coronapandemie fünf Spieler Auswechslungen erlaubt, diese dürfen während der Spielzeit in maximal drei Unterbrechungen sowie in der Halbzeitpause durchgeführt werden. Die Spielzeit bleibt bei einem Spiel nie stehen, sie läuft kontinuierlich weiter, unerheblich davon welches spielerische Ereignis geschieht (Tor, Abseits, Foul, Einwurf, Auswechslung usw.).

Während eines Spiels gibt es pro Mannschaft elf verschiedene Spieler. Mit Ausnahme von dem Torwart werden alle Spieler als Feldspieler angesehen. Als Startelf werden die elf Spieler angesehen, welche zu Beginn des Spiels auf dem Spielfeld stehen, also das Spiel beginnen. Als Ersatzspieler oder Auswechselspieler werden die, sieben bis neun, Spieler angesehen, welche

zu Beginn des Spiels nicht auf dem Feld stehen, sondern auf der Ersatzbank sitzen, und während des Spiels eingewechselt werden können.

Unter den Spielern der Startelf gibt es verschiedene Positionen. Als Abwehrspieler oder Verteidiger werden die Feldspieler angesehen, welche am nächsten an dem eigenen Torwart positioniert sind, ihre Aufgabe ist primär das Verhindern von Gegentoren. Üblich sind zwischen drei und fünf Abwehrspieler. Unter den Abwehrspielern wird zwischen den Innenverteidigern und den Außenverteidigern unterschieden, wobei als Außenverteidiger die zwei Feldspieler angesehen werden, welche am Rand des Spielfeldes positioniert sind und als Innenverteidiger diejenigen die im inneren des Spielfeldes positioniert sind. Als Mittelfeldspieler werden die Feldspieler angesehen, welche vor den Abwehrspielern, im Mittelraum des Spielfeldes, positioniert sind. Die Mittelfeldspieler werden weiter in Offensive, Defensive, Zentrale und Außenmittelfeldspieler unterteilt. Als Stürmer werden die Feldspieler angesehen, die als primäre Aufgabe das Erzielen von Toren haben, sie sind üblicherweise am nächsten an dem gegnerischen Tor positioniert.

In dieser Arbeit werden vor allem die Verteidiger der Mannschaft des HSV aufgezeichnet, später jedoch auch Spieler anderer Mannschafften.

2.2 Aufbau der *dfldata*

Die einzelnen Spieler sind in der MongoDB Database *dfldata* abgespeichert. Hier findet man sie in der Collection *players*. Die Spieler sind als Dictionarys mit einem Namen (der volle Name), einem Team-Namen, einem Nachnamen, einer Spieler-ID und einem Array an Werten der verschiedenen Attribute an den unterschiedlichen Spieltagen, abgespeichert. Dieses Array (*statistics*) besteht aus 34 Werten (0-33), welche die Spieltage repräsentieren. Innerhalb dieser Spieltags-Objekte sind die neun Attribute, die abgefragt werden sollen, eine boolesche Variable die darüber Auskunft gibt, ob der Spieler an jenem Tag spielte (*hasPlayed*), und eine Variable die den Spieltag als Zahl angibt (*matchDay*).

Abbildung 1: Aufbau eines Spieler-Dictionarys unter *statistics* (Datenbank *dfldata*)

Falls der Spieler an einem Spieltag nie spielte, dann enthält der Wert des *statistics* Arrays an diesem Punkt *null*. Die Datenbank gibt keine Auskunft darüber, über welch Saison es sich bei einem Spieltag handelt. Dies bedeutet, dass beispielsweise am Index 5 der *statistics* der sechste Spieltag der Saison 2019/2020 enthalten sein könnte an Index 6 jedoch der siebte Spieltag der Saison 2020/2021. Der Inhalt hängt von der abgefragten URL ab, welche für das Füllen der Datenbank zuständig ist und in einem, vom HSV übergebenen Programm, abgeändert wird. Der Wert eines Spieltages in *statistics* wird nur überschrieben, wenn der Spieler in der aktuell abgefragten Saison erneut an diesem Spieltag zum Einsatz kommt. Falls er in der aktuell abgefragten Saison nicht an diesem Spieltag spielte, so bleibt sein ursprünglicher Wert der letzten abgefragten Saison bestehen. Daher kann es dazu kommen, dass ein Spieler der, an dem aktuell abgefragtem Spieltag der Saison 2020/2021, nicht gespielt hat trotzdem einen Wert unter dem dazugehörigen Index in dem *statistics* Array hat, da er an diesem Spieltag der vergangenen Saison 2019/2020 gespielt hatte. Voraussetzung dafür ist nur, dass dieser Spieltag der Saison 2019/2020 vorher durch eine entsprechende URL abgefragt wurde, die Datenbank daher befüllt wurde. Dementsprechend kann der Wert in *statistics* nur *null* sein, wenn der Spieler noch in keiner der abgefragten Saisons an diesem Spieltag spielte.

2.1.1 Aufgezeichnete Attribute

Die hier aufgezeichneten Attribute sind wie folgt im Definitionskatalog der offiziellen Spieldaten (DFL Deutsche Fußball Liga GmbH, 2018/2019) definiert:

- assistsShotsAtGoal: Gegeben durch die Anzahl aller direkten, sonstigen und Freistoß-Torvorlagen
    - direkte Torvorlage: Ballaktion innerhalb einer kurzen Zeitspanne vor dem Spielereignis Torschuss, bei der der Torschussschütze Empfänger des Balls ist.
    - Sonstige Torvorlage: Liegt vor, wenn der Ball zwar den Torschützen von einem Mitspieler aus erreicht, diese Aktion aber nicht primär die Intention einer Torvorlage hat.
    - Freistoß-Torvorlage: Liegt vor, wenn der Ball den Torschussschützen bei einem Freistoß als Freistoßablage erreicht.
- averageSpeed: Das Durchschnittstempo eines Spielers ist gegeben durch den Quotienten aus der Gesamtlaufleistung dieses Spielers und seiner Einsatzzeit.
- completedPasses: Gegeben durch die Anzahl aller Pässe aus dem Spiel abzüglich der Fehlpässe.
    - Pässe: Gegeben durch die Anzahl aller Pässe aus Standartaktionen und Pässe aus dem Spiel (ohne Pässe aus Einwürfen)
        - Standartaktionen: Anstoß, Abstoß, Einwurf, Eckstoß, Freistoß und Strafstoß
    - Fehlpässe: Gegeben durch die Anzahl der nicht erfolgreichen Pässe aus dem Spiel
- distance: Die Laufdistanz eines Spielers zwischen zwei Frames ist gegeben durch die Angabe des Distanzwertes im zweiten Frame
- sprints: Ein Sprint liegt vor, wenn ein Spieler mehr als zwei Sekunden mindestens 4,0 m/s läuft und während dieser Zeit mindestens 6,3 m/s läuft, wobei zwischen dem ersten und letzten Erreichen der 6,3 m/s Schwelle mindestens eine Sekunde liegt. Der Sprint ist zeitlich auf die Zeitspanne begrenzt, in der die 6,3 m/s Schwelle das erste und das letzte Mal erreicht wird. Ein neuer Sprint kann erst dann vorliegen, wenn die Schwelle von 4,0 m/s unterschritten wird.
- intenseRuns: Die Anzahl der Intensiven Läufe ist gegeben über die Anzahl aller Tempoläufe und Sprints

  - Tempoläufe: Ein Tempolauf liegt vor, wenn ein Spieler mehr als zwei Sekunden mindestens 4,0 m/s läuft und während dieser Zeit mindestens 5,0 m/s läuft, wobei zwischen dem ersten und dem letzten Erreichen der 5,0 m/s Schwelle mindestens eine Sekunde liegt. Gleichzeitig darf die Sprintdefinition nicht erfüllt sein. Der Tempolauf ist zeitlich auf die Zeitspanne begrenzt, in der die 5,0 m/s Schwelle das erste und das letzte Mal erreicht wird. Ein neuer Tempolauf kann erst dann vorliegen, wenn die Schwelle von 4,0 m/s unterschritten wird.
- maxSpeed: Das Maximaltempo eines Spielers ist gegeben über den Maximalwert aller Tempoläufe für diesen Spieler.
- shotsAtGoal: Gegeben durch die Anzahl aller Erfolgreichen Torschüsse (Tore), Gehaltenen Torschüsse, Torschüsse an den Rahmen und aller Geblockten Torschüsse bei denen ein Tor verhindert wurde

Das Attribut *passesLastThird* wurde nicht explizit im Definitionskatalog (DFL Deutsche Fußball Liga GmbH, 2018/2019) gelistet und kann daher nicht beschrieben werden, trotzdem wurde auch dieses Attribut aufgezeichnet.

Hinweis zur Berechnung der Attribute aus dfldata

Abgesehen von den Attributen *passesLastThird*, *averageSpeed* und *maxSpeed* werden alle anderen Werte der Attribute berechnet, indem die von der DFL gelieferten Rohwerte durch die Spielzeit des Spielers geteilt werden.

2.3 Datenbeschaffung/Implementierung

2.3.1 Spieltag Abfrage

Für die Abfrage über die Spieltage wurde zunächst die Datenbank *dfldata* gefüllt. Dafür wurde das vom HSV bereitgestellte Programm genutzt. In diesem wurden die Parameter *MATCH_URL*, *MATCH_URL_POSITION* und *MATCH_DAY* für jeden Spieltag angepasst. Die ersten zwei Parameter sind URLs und beinhalten eine Variable für die gewollte Saison und eine Variable für den gewollten Spieltag. Die Variable für die Saison 2019/2020 lautet *DFL-*

SEA-0001K3 und die für die Saison 2020/2021 *DFL-SEA-0001K4*. Die Variablen für die einzelnen Spieltage unterscheiden sich in ähnlicher Art und Weise. Der Parameter *MATCH_DAY* besteht aus einer Zahl zwischen eins und 34. Nachdem die Datenbank befüllt war, wurde ein Programm *reader.py* geschrieben, welches die Daten ausliest und in einer CSV-Datei abspeichert. Die Datei hatte folgendes Format: *Nachname des Spielers, Spieltag, <Attribut_1>, <Attribut_2>, …, <Attribut_8>*. Dabei wurden nur Spieler beachtet, welche im Kader des HSV als Verteidiger gelistet wurden.

reader.py

connect(): Diese Methode erstellt einen Mongo-Client und ruft auf diesem den gewünschten Unterordner auf, welcher die Spielerdaten beinhaltet, in diesem Fall *players*. Rückgabewert ist eine Collection an Spieler-Dictionarys.

nurVerteidiger(players): Die Methode bekommt eine Collection an Spieler-Dictionarys *players* übergeben und filtert die Verteidiger aus diesem heraus. Rückgabewert ist eine Liste der Nachnamen der Verteidiger.

playerStats(playerName): Bekommt einen String *playerName* übergeben. Sucht in der Collection *players* nach dem Eintrag für den Spieler *playerName* und sammelt alle vorhandenen Werte unter dem Punkt *statistics* in einem Array *spieltagArray*. Die in *spieltagArray* enthaltenen Einträge sind Dictionarys, von diesen werden dann jeweils die Einträge unter *_id* gelöscht. Gibt *spieltagArray* zurück.

allInOneCSV(playerNames): Bekommt eine Liste an Spieler-Nachnamen übergeben und erstellt daraufhin eine CSV-Datei des oben angegebenen Formats welche alle Spieler, die in *playerNames* enthalten sind, beinhaltet.

2.3.2 Spielminute Abfrage

Für die Abfrage der Attribute pro Spielminute muss jede Minute eines laufenden Fußballspiels die Datenbank *dfldata* abgefragt werden. Dafür muss zunächst vor jedem Spiel das vom HSV zur Verfügung gestellte Programm mit den entsprechenden Parametern für *MATCH_URL*, *MATCH_URL_POSITION* und *MATCH_DAY* gestartet werden. Es wurden insgesamt vier

verschiedene Programme geschrieben, welche zur Abfrage genutzt werden können. Zwei von ihnen sind für die Aufzeichnung der ersten Halbzeit zuständig (*HZ1min.py* und *20Sek_HZ1.py*), die anderen beiden sind für die Aufzeichnung der zweiten Halbzeit zuständig (*HZ2min.py* und *20Sek_HZ2.py*). Durch die fehlenden Informationen bezüglich der aktuellen Spielminute ist es wegen der Halbzeitpause nicht möglich beide Halbzeiten in einem Programm abzufragen, ohne eine feste Zeitangabe für die Halbzeitpause zu machen, welche zu zeitlich verfälschten Daten führen kann. Die Programme müssen daher bei Beginn der Halbzeiten manuell gestartet werden und laufen durch einen implementierten Zähler je 45 Minuten lang. Der Unterschied zwischen den Programmen *HZ1min.py* und *20Sek_HZ1.py* sowie zwischen *HZ2min.py* und *20Sek_HZ2.py* besteht in der Frequenz der Abfragen. Die Programme *HZ1min.py* und *HZ2min.py* fragen die *dfldata* pro Minute einmal ab, während die Programme *20Sek_HZ1.py* und *20Sek_HZ2.py* pro Minute drei Abfragen in abständen von 20 Sekunden tätigen. Alle Programme speichern die abgerufenen Werte der Attribute pro Spieler in einer CSV-Datei des Formats: *Zeit, <Attribut_1>, <Attribut_2>, ..., <Attribut_8>*. Wobei auch hier ein kleiner Unterschied besteht, denn die Programme *HZ1min.py* und *HZ2min.py* speichern die Zeit nicht in der Spalte namens *Zeit*, sondern in einer Spalte namens *Minute* ab. Bei beiden Varianten sind die Programme der ersten Halbzeit für die Erstellung der CSV-Dateien zuständig, während die Programme der zweiten Halbzeit nur Zeilen an die schon erstellten CSV-Dateien anfügen.

Die Programme fragen dabei alle Werte ab, welche in der *dfldata* unter dem gewünschten Spieltag zu finden sind. Dies bedeutet, dass sie auch Werte von Spielern aufzeichnen können, welche nicht der aktuellen Saison entsprechen, sondern einer vorherigen Abfrage einer anderen Saison. Aus den erstellten CSV-Dateien wird dann ersichtlich welche Spieler tatsächlich an dem Spieltag der gewünschten Saison spielten und welche nicht. Denn die Werte der Spieler die tatsächlich gespielt haben ändern sich von Zeile zu Zeile, während die Werte jener Spieler, die in einer anderen Saison an diesem Spieltag spielten, konstant gleichbleiben. Spieler, die ausgewechselt wurden, sind daran zu erkennen, dass ihre Werte ab einer bestimmten Minute gleichbleiben. Die CSV-Dateien von Spielern, die eingewechselt wurden, beginnen entweder erst ab der Minute der Einwechslung, falls vorher keine Werte in der Datenbank enthalten waren, oder enthalten konstant gleichbleibende Werte bis zur Minute der Einwechslung, ab welcher sich die Werte dann voneinander unterscheiden.

Da die Programme von Halbzeit eins und Halbzeit zwei sich nur minimal voneinander unterscheiden werden hier nur die Programme der ersten Halbzeiten gelistet. Diese enthalten alle Methoden, welche auch in den Programmen der zweiten Halbzeit enthalten sind.

20Sek_HZ1.py

playerStats(players, playerName, spieltag): Bekommt eine Collection von Spieler-Dictionarys *players*, einen String *playerName* und eine Zahl *spieltag* übergeben. Sucht in der Collection *players* nach dem Eintrag für den Spieler *playerName* und sammelt die Werte unter dem Punkt *statistics*, bei welchem der *matchDay* dem übergebenen *spieltag* entspricht, in einer Liste *spieltagArray*. Gibt *spieltagArray* zurück.

initiale_csv(verteidiger, players, zeit matchDay): Bekommt eine Liste mit den Nachnamen der Verteidiger *verteidiger*, eine Collection von Spieler-Dictionarys *players* und eine Zahl *spieltag* übergeben. Erstellt für jeden Spieler, der in *verteidiger* enthalten ist, eine CSV-Datei des Formats: *Zeit, <Attrinut_1>, <Attribut_2>, ..., <Attribut_8>*. Der Name der CSV-Dateien stellt sich jeweils aus dem Nachnamen des Spielers sowie dem *spieltag* zusammen.

csv_fuellen(verteidiger, players, matchDay): Bekommt eine Liste mit den Nachnamen der Verteidiger *verteidiger*, eine Collection von Spieler-Dictionarys *players*, eine Zahl *zeit* und eine Zahl *spieltag* übergeben. Für jedes Dictionary in *players* bei welchem der Eintrag *lastName* in *verteidiger* enthalten ist wird zunächst der Rückgabewert der Methode *playerStats()*, mit den Parametern *players*, *spieltag* und dem jeweiligen Element aus *verteidiger*, in der Variablen *playerArray* gespeichert. Füllt daraufhin die zum *playerName* zugehörige CSV-Datei mit den Inhalten der Werte, die in *playerArray* enthalten sind.

min_abfrage(verteidiger, spieltag): Bekommt eine Liste gefüllt mit Stings *verteidiger* und eine Zahl *spieltag* übergeben. Fragt die Datenbank *dfldata* drei Mal pro Minute ab, wobei zwischen jeder Abfrage 20 Sekunden gewartet wird. Zählt mit einem Counter *i* die Minuten und mit einem Counter *sleeper* die Sekunden mit. Unterscheidet zwischen der ersten Aufzeichnung (Minute eins und Sekunde null) eines Spiels, bei welcher noch CSV-Dateien erstellt werden müssen, und den restlichen Minuten, bei welchen nur noch in die bereits erstellten Dateien geschrieben werden muss. Falls es sich um die erste Aufzeichnung eines Spiels handelt wird die Methode *initiale_csv()* aufgerufen und danach die Methode *csv_fuellen()*. Falls es nicht die

erste Aufzeichnung ist so wird nur die Methode *csv_fullen()* aufgerufen. Eine boolesche variable *first* wird verwendet, um festzuhalten, ob es sich um die erst Aufzeichnung handelt.

HZ1min.py

Die Methode *playerStats()* ist identisch wie bei *20Sek_HZ1.py*. Die Methoden *initiale_csv()* und *csv_fuellen()* unterscheiden sich nur durch die Namensgebung der ersten Spalte der CSV-Dateien. Hier ist das Format dieser wie folgt: *Minute, <Attrinut_1>, <Attribut_2>, ..., <Attribut_8>*.

min_abfrage(verteidiger, spieltag): Bekommt eine Liste gefüllt mit Stings *verteidiger* und eine Zahl *spieltag* übergeben. Fragt die Datenbank *dfldata* jede Minute einmal ab, wobei zwischen jeder Abfrage 60 Sekunden gewartet wird. Zählt mit einem Counter *i* die Minuten mit. Unterscheidet zwischen der ersten Aufzeichnung (Minute eins) eines Spiels, bei welcher noch CSV-Dateien erstellt werden müssen, und den restlichen Minuten, bei welchen nur noch in die bereits erstellten Dateien geschrieben werden muss. Falls es sich um die erste Aufzeichnung eines Spiels handelt wird die Methode *initiale_csv()* aufgerufen und danach die Methode *csv_fuellen()*. Falls es nicht die erste Aufzeichnung ist so wird nur die Methode *csv_fullen()* aufgerufen.

2.4 Explorative Analyse der Daten

2.4.1 Spieltage

Um ein besseres Verständnis für die Daten zu bekommen, wurde eine explorative Analyse der Daten vorgenommen, beginnend mit den Daten, welche aus den Spieltagen der Saison 2019/2020 gesammelt wurden. Diese wurden mit dem Programm *reader.py* erstellt und beinhalteten 188 Zeilen und maximal 34 Werte pro Spieler, bei sechs verschiedenen Verteidigern. Sie lagen in der folgenden Form vor: *Spielername, Spieltag, <Attribut_1>, <Attribut_2>, ..., <Attribut_8>*. Es wurde erkenntlich, dass einige Spieler deutlich mehr Einsätze in dieser Saison hatten als andere. Dementsprechend schwierig würde es sein, für einige Spieler das

geplante Clustering durchzuführen, da diese schlichtweg zu wenig Einsätze hatten.

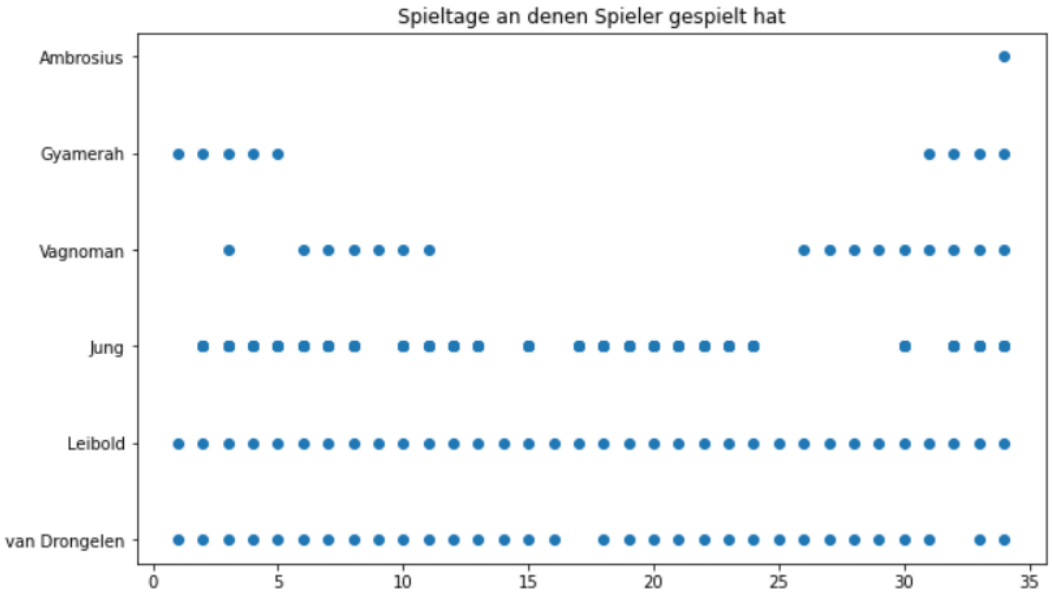

Abbildung 2: Plot der Häufigkeit der Spieleinsätze der Verteidiger (Eigendarstellung)

Es wurde außerdem die Anzahl an Nullen gezählt, die pro Attribut vorkommen. Es zeigte sich, dass einige Attribute keine Nullwerte vorweisen (*sprints, averageSpeed, distance und intenseRuns*) andere jedoch sehr viele (*shotsAtGoal und assistsShotsAtGoal*). Die Attribute *shotsAtGoal* und *assistsShotsAtGoal* wiesen von 188 Werten 119 und 117 Nullwerte auf, was 62% bzw. 63% der gesamten Daten ausmacht. Das Attribut *passesLastThird* wies 52 Nullwerte auf. Da die aufgezeichneten Werte zu Verteidigern gehören ist es nicht sonderlich überraschend, dass diese Attribute eine große Anzahl an nullen aufweisen und es nur wenige Werte gibt, die darüber hinaus gehen. Durch Boxplots wurden die Wertebereiche der verschiedenen Attribute noch genauer beleuchtet. Bei den Attributen *shotsAtGoal* und *assistsShotsAtGoal* handelt es sich um einen Wertebereich von null bis ungefähr 0.2. Der Großteil der Werte beider Attribute ist wie schon erwähnt null und bei beiden werden Werte über 0.02 von dem Boxplot als Ausreißer angesehen. Bei den *shotsAtGoal* gibt es drei solcher Ausreißer, von denen der Größte bei über 0.1 liegt. Bei *assistsShotsAtGoal* gibt es vier Ausreißer, von denen der Größte bei 0.1 liegt. Die restlichen Attribute zeigen eine größere Variation der Werte auf, die nicht als Ausreißer angesehen werden. Die Wertebereiche unterscheiden sich hierbei von Attribut zu Attribut. Die Boxplots machen klar, dass es Werte gibt welche als Ausreißer angesehen werden

können, ob tatsächlich eine Ausreißerbehandlung notwendig ist wird jedoch noch nicht festgelegt. Stattdessen wird auf die ersten Clustering Ergebnisse und weitere Daten gewartet.

Um das Verhältnis der Attribute untereinander zu beleuchten, wurde eine Heatmap erzeugt.

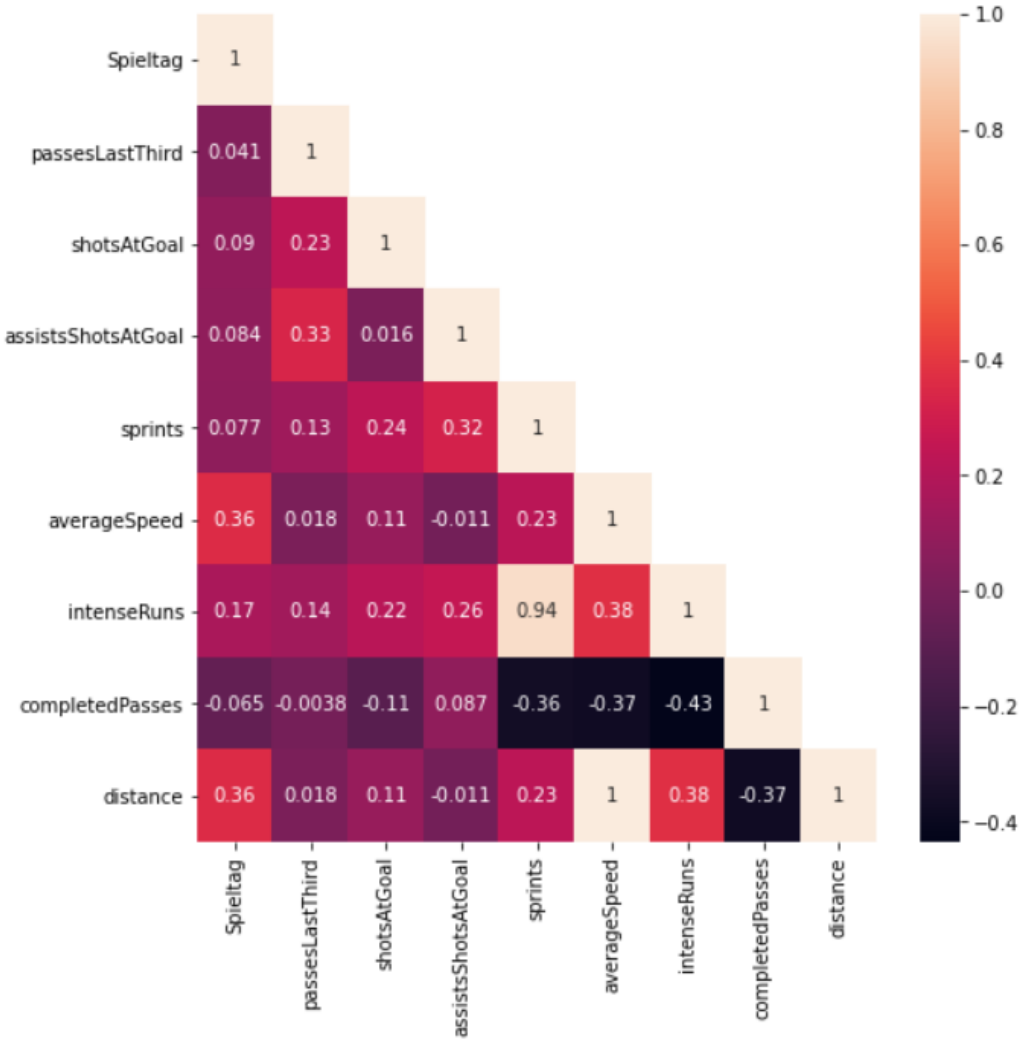

Abbildung 3: Heatmap der Attibute aus *dfldata* (Eigendarstellung)

Diese zeigt deutlich, dass die Attribute *averageSpeed* und *distance* stark voneinander abhängen. Dies ist wenig überraschend, da, um mehr Distanz in der gleichen Zeit zu überbrücken, schneller gelaufen werden muss und damit die Durchschnittsgeschwindigkeit steigt. Die Attribute *sprints* und *intenseRuns* hängen ebenfalls stark voneinander ab. Dies war ebenfalls zu erwarten, denn der Wert für das Attribut *intenseRuns* beinhaltet in seiner Berechnung den Wert

des Attributs *sprints*. Das Attribut *completedPasses* weist die meisten negativen Abhängigkeiten auf. Dies könnte dadurch zu erklären sein, dass für abgeschlossene Pässe eine erhöhte Geschwindigkeit, die mit den Attributen *sprints*, *intenseRuns* und *averageSpeed* einhergeht, kontraproduktiv ist. Die Attribute *completedPasses* und *distance* weisen dementsprechend auch eine negative Abhängigkeit auf.

2.4.2 Minuten

Auch hier wurde eine explorative Analyse der Daten vorgenommen, diese stammen vom 26. Spieltag der Saison 2020/2021. Dieser Datensatz entstand durch die Programme, welche die *dfldata* drei Mal pro Minute abfragen und ist zusammengestellt aus den vier Datensätzen der vier Verteidiger, die an diesem Spieltag spielten und enthält insgesamt 1080 Zeilen. Die Datensätze wurden hierbei um die zwei Spalten „Name“ und „Rolle“ erweitert, die Spalte „Name“ enthält den Nachnamen des jeweiligen Spielers, während die Spalte „Rolle“ sich darauf bezieht, ob der Spieler als Außen- oder Innenverteidiger tätig war. Auch hier wurde klar, dass vor allem die Attribute *shotsAtGoal*, *assistsShotsAtGoal* aber auch *passesLastThird* viele

Nullwerte aufweisen. Diese Attribute würden sich daher weniger für das Clustering eignen.

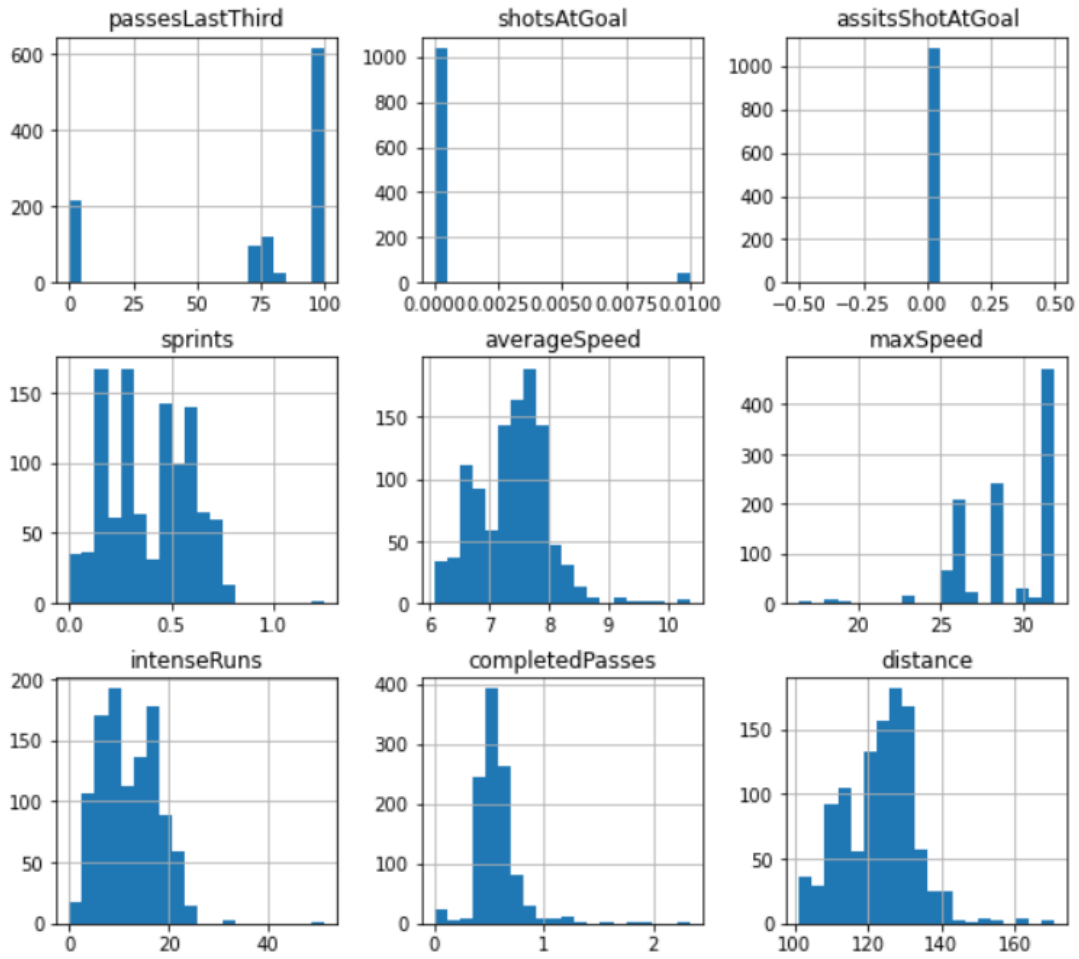

Abbildung 4: Plotten der Werte der Attribute aus *dfldata* (Eigendarstellung)

Bei den betroffenen Attributen handelt es sich um solche, die offensive Aktionen beschreiben, für diese sind die Verteidiger nicht primär zuständig. Daher und da die Ergebnisse des Clusterings mit dieser Art von ständig gleichbleibenden Werten nicht gut funktionieren würde, sollten diese Attribute vom Clustering ausgeschlossen werden. Die übrigen Attribute zeigen eine stärkere Verteilung der vorkommenden Werte auf, würden sich daher besser für das Clustering eignen.

Die Analyse zeigte des Weiteren, dass zu Beginn des Spiels die Werte der Attribute *averageSpeed*, *distance*, *completedPasses* und *sprints* ihr Maximum erreichen.

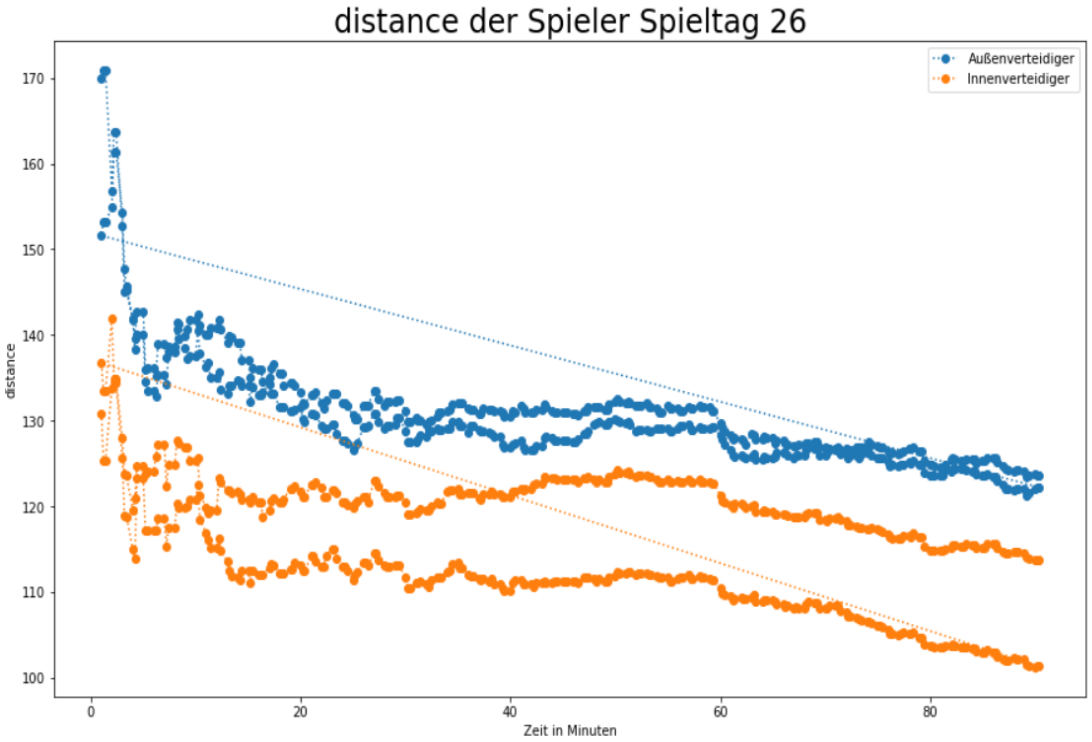

Abbildung 5: *distance* der Spieler am Spieltag 26. nach Rollen (Eigendarstellung)

Allerdings handelt es sich hierbei, ausgenommen von *averageSpeed*, um Attribute, die mit der Spielminute verrechnet werden und da am Anfang durch eine kleinere Zeit geteilt wird, war zu erwarten, dass hier die Werte erhöht sein werden. Außerdem wurde klar, dass es Unterschiede zwischen den Innenverteidigern und den Außenverteidigern gibt. Die Außenverteidiger sprinten mehr und legen eine höhere Distanz zurück als die Innenverteidiger, dementsprechend liegt auch ihre Durchschnittsgeschwindigkeit über jener der Innenverteidiger. Die Innenverteidiger hingegen weisen mehr vollendete Pässe auf als die Außenverteidiger. Es ist in fast allen Attributen zu beobachten, dass die Werte der Spieler im Laufe des Spiels tendenziell abnehmen.

Dies wurde mit den Daten vom Spieltag 28 wiederholt, dieser Datensatz bestand aus 1081 Zeilen und wurde ebenfalls durch die Programme der dreimaligen Abfrage pro Minute ertsellt. Hier zeigten sich in der zweiten Halbzeit des Spiels in den Attributen *distance*, *averageSpeed*, *completedPasses* und *shotsAtGoal* auch ungewöhnlich hohe Werte.

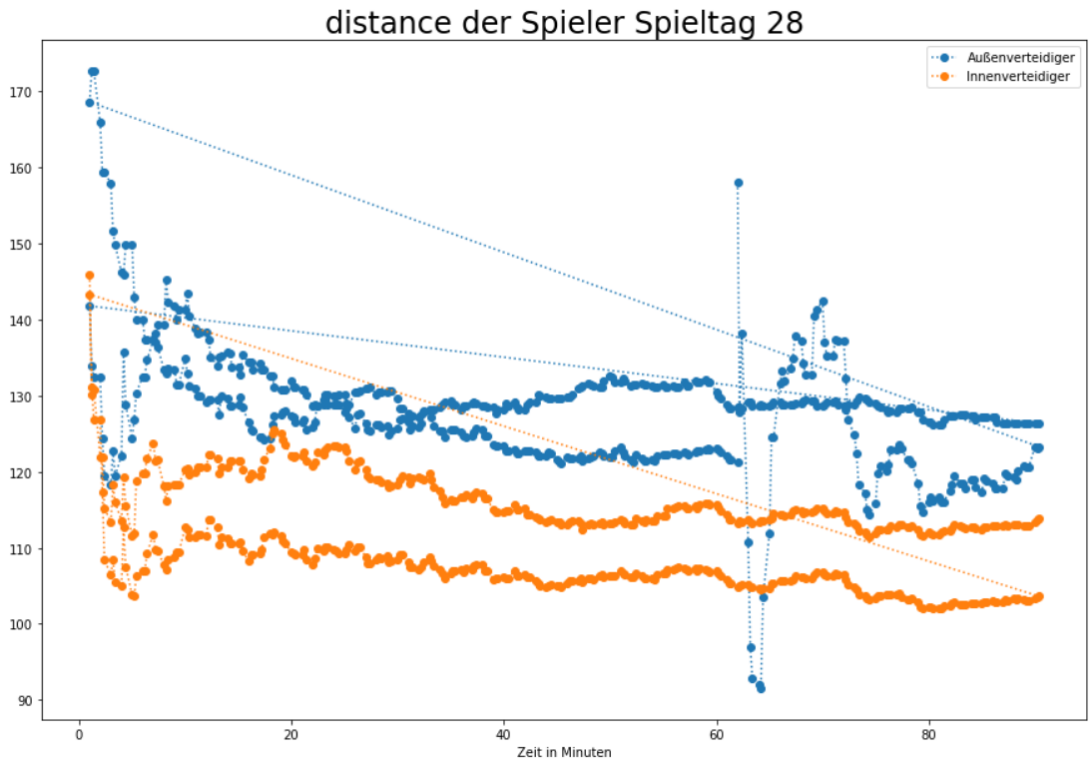

Abbildung 6: *distance* der Spieler an Spieltag 28. nach Rollen (Eigendarstellung)

Hier zu sehen sind die stark abweichenden Werte ab der 60. Minute, diese beginnen deutlich über den anderen Werten, fallen dann stark ab, um wieder erneut anzusteigen. Allgemein ist in diesen Werten eine stärkere Varianz zu erkennen, ihre Werte unterscheiden sich von Minute zu Minute deutlich stärker als es die Werte der anderen Verteidiger tun.

Nach Umstellung der farblichen Unterscheidung die Namen der Spieler statt ihren Rollen stellte sich heraus, dass diese Werte zu dem Spieler *Gyamerah* gehörten, dieser wurde in der

61. Minute des Spiels eingewechselt.

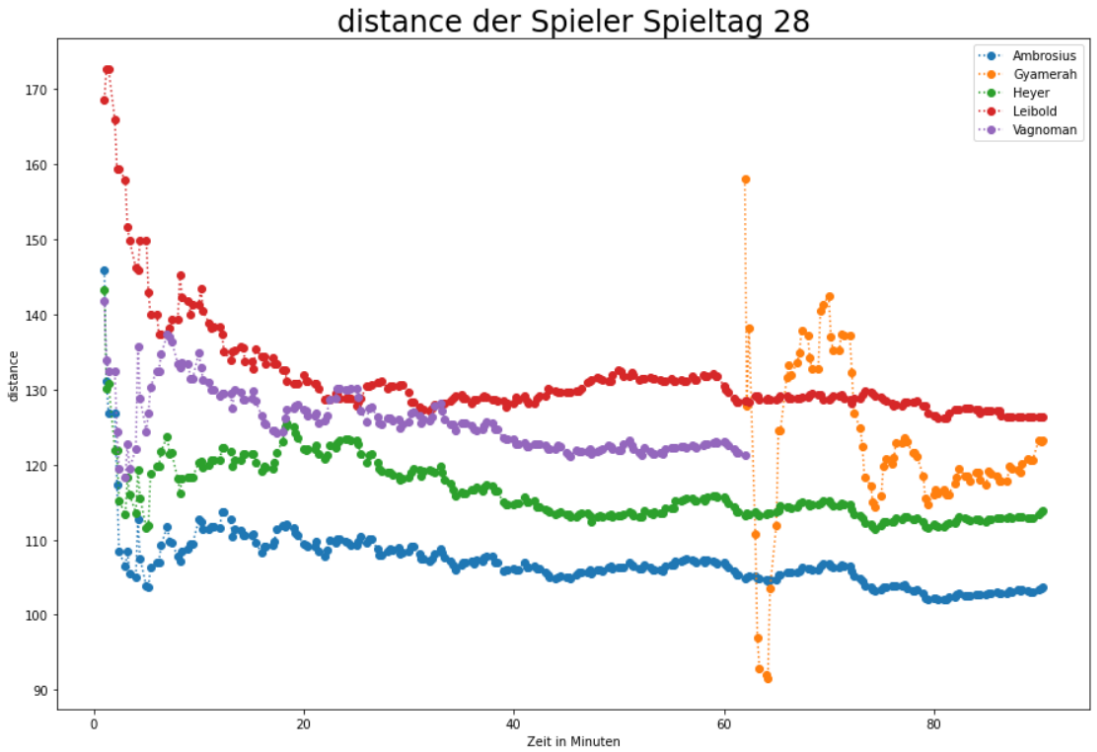

Abbildung 7: *distance* der Spiler an Spieltag 28. nach Namen (Eigendarstellung)

Es ist daher logisch, dass seine Werte sich stärker von denen der restlichen Spieler unterscheiden, da diese Attribute mit der Spielzeit des jeweiligen Spielers verrechnet werden. Somit hat *Gyamerah* in der 61. Minute erst eine Spielzeit von einer Minute, währen die anderen Spieler eine Spielzeit von 61. Minuten aufweisen. Durch die Verrechnung mit der Spielzeit entstehen dementsprechend bei *Gyamerah* stärkere Abweichungen als bei den anderen Spielern. Somit stellen diese Werte logisch keine Ausreißer dar, es stellt sich jedoch die Frage, ob sie aufgrund der starken Abweichung zu den anderen Werten, nicht doch als solche anzusehen sind. Da das Clustering und die Klassifikation pro Spieler erstellt werden sollen, werden diese Werte schlussendlich nicht als Ausreißer angesehen werden. Denn sie könnten für den jeweiligen Spieler wichtig sein, wenn dieser beispielsweise oft spät eingewechselt wird.

2.5 Die Zweite Dimension

Da die Daten aus der Datenbank *dfldata* eindimensional sind, jedoch für die Clusterung zweidimensionale Daten benötigt werden stellte sich die Frage, welcher Wert dazu benutzt werden sollte. Dabei ergaben sich mit den Minuten und den Spieltagen zwei Möglichkeiten.

2.5.1 Spieltage

Durch die explorative Analyse wurde die Vermutung nahegelegt, dass sich die Spieltage nicht gut für das Clustern der Daten eignen. Ein Grund dafür liegt in der Unterschiedlichen Menge an Daten, die pro Spieler zur Verfügung stehen. Dies macht das Clustern eines Spielers, der nur ein Spiel in der Saison hatte, sinnlos, denn er hat für jedes Attribut nur einen Wert zur Verfügung. Allgemein liegt die Vermutung nahe, dass selbst wenn ein Spieler an jedem Spieltag der Saison spielte, noch zu wenig Daten zur Verfügung stehen würden, um ein sinnvolles Ergebnis zu erhalten. Denn eine Saison besteht aus 34 Spieltagen also 34 Werte die maximal pro Spieler und pro Attribut genutzt werden können. Um dies zu veranschaulichen, wurde

beispielhaft mit KMeans das Attribut *sprints* von

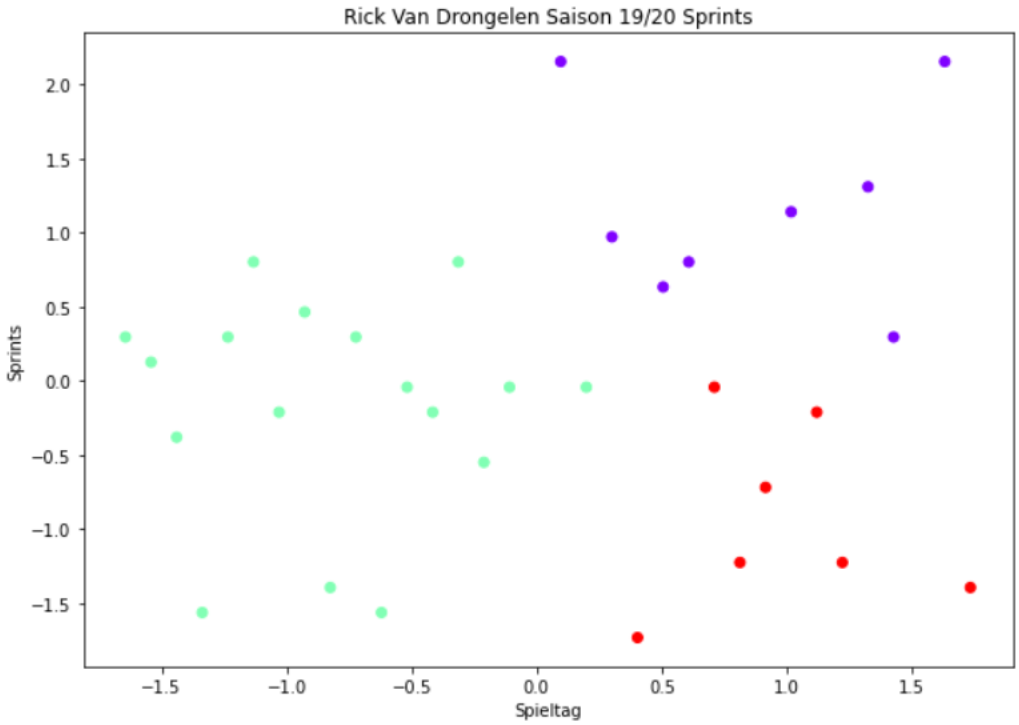

Abbildung 8: KMeans Clustering von *sprints* von Van Drongelen Saison 19/20 mit drei Clustern (Eigendarstellung)

dem Spieler *Rick van Drongelen* geclustert, dieser hatte in der Saison 2019/2020 32 Einsätze. Dabei wurden die Daten zuerst mit dem *StandardScaler* skaliert. Durch die Ergebnisse stellte sich die Frage, ob es sinnvoll wäre einen Wert an einem Spieltag als „Gut“ zu klassifizieren und an einem anderen Spieltag als „Schlecht“, was hier der Fall wäre. Die Leistungen von Spielern können, bei von den Spielern als „wichtiger“ empfundenen Spielen, zwar ansteigen, jedoch wird dieser Effekt nicht in der 2 Bundesliga beobachtet (Link & de Lorenzo, 2016). Außerdem sind solche Spiele nicht immer an denselben Spieltagen gelegen. Daher lässt sich sagen, dass sich die Spieltage nicht als zweite Dimension zum Clustern eignen.

2.5.2 Minuten

Nach Absprache mit dem HSV bezüglich der zweiten Dimension wurde die Spielminute als zweite Dimension empfohlen. Dafür mussten die Attribute, während der Live-Spiele, minütlich abgefragt werden. Damit stünden pro Spiel für jeden aufgezeichneten Spieler maximal 90

Werte pro Attribut zur Verfügung und dementsprechend maximal $90 \times 34 = 3060$ Werte pro Attribut für eine Saison. Durch die spätere Implementierung einer Abfrage, welche pro Minute die Datenbank *dfldata* drei Mal abfragt, würden sogar 9180 Werte pro Saison zur Verfügung stehen können. Auch macht die Spielminute als zweite Dimension mehr Sinn bezüglich des Clusterings, da beispielsweise die Werte von Attributen wie *sprints*, *distance* und *intenseRuns* in der zweiten Hälfte des Spiels abnehmen. Allgemein nehmen die Leistungen der Spieler mit zunehmender Spielminute eher ab (Armatas, Yiannakos, & Sileloglou, 2007). Daher müssten hier unterschiedliche Maßstäbe für die Werte der verschiedenen Minuten gesetzt werden. Diese Abnahme der Leistungen war auch in der explorativen Analyse zu beobachten.

2.6 Clustering

Nachdem beschlossen wurde mit den Minuten als zweite Dimension zu arbeiten begannen die ersten Clustering-Versuche. Die erste Aufzeichnung war dabei die des 24. Spieltages der Saison 2020/2021, hierbei wurden nur die Werte der vier Verteidiger des HSV aufgezeichnet und die Datenbank wurde einmal pro Minute abgefragt. Für den Anfang wurde beispielhaft KMeans für das Clustering genutzt. Durch die Elbow-Methode wurde die ideale Anzahl von zwei Clustern bestimmt.

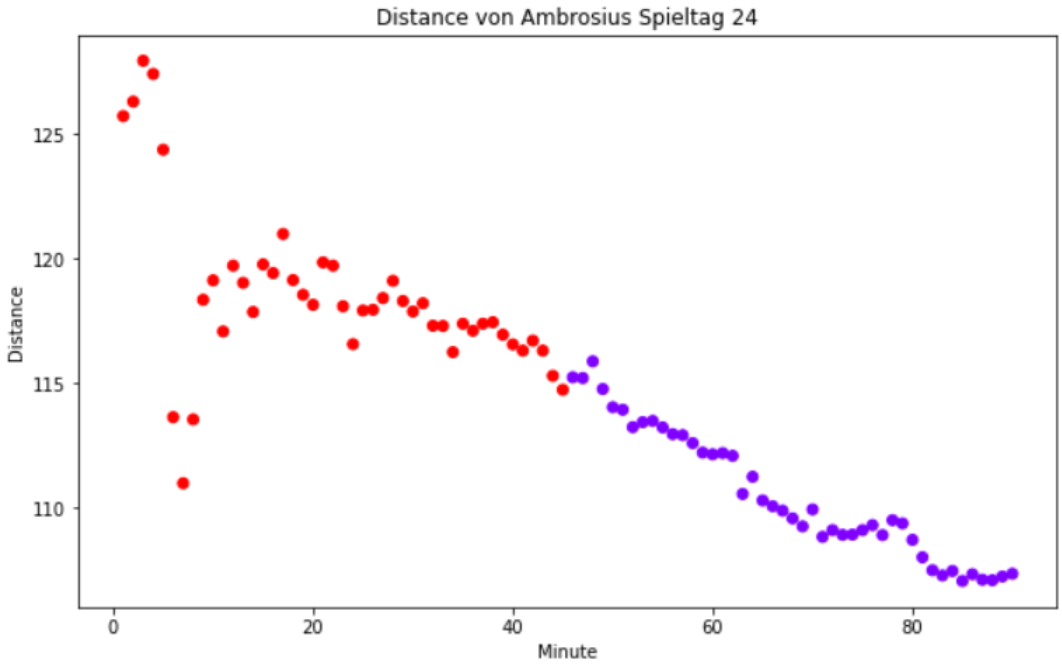

Abbildung 9: KMeans Clustering von *distance* von Ambrosius Spieltag 24. (Eigendarstellung)

Das Ergebnis lieferte nicht die gewünschte Unterscheidung zwischen „Guten“ und „Schlechten“ Werten, sondern lediglich eine Trennung der Cluster abhängig von der Spielminute. Allerdings war aufgrund der geringen Menge an Daten nicht mit einem vernünftigen Ergebnis zu rechnen. Daher wurden alle vier aufgezeichneten Verteidiger beispielhaft in ein Dataframe verpackt, um mehr Daten zu erhalten, ohne einen weiteren Spieltag abwarten zu müssen. Dieses Mal wurden die Daten mithilfe des *StandardScalers* skaliert und erneut mit KMeans geclustert, laut Elbow-Methode war hierbei eine Anzahl von drei Clustern optimal.

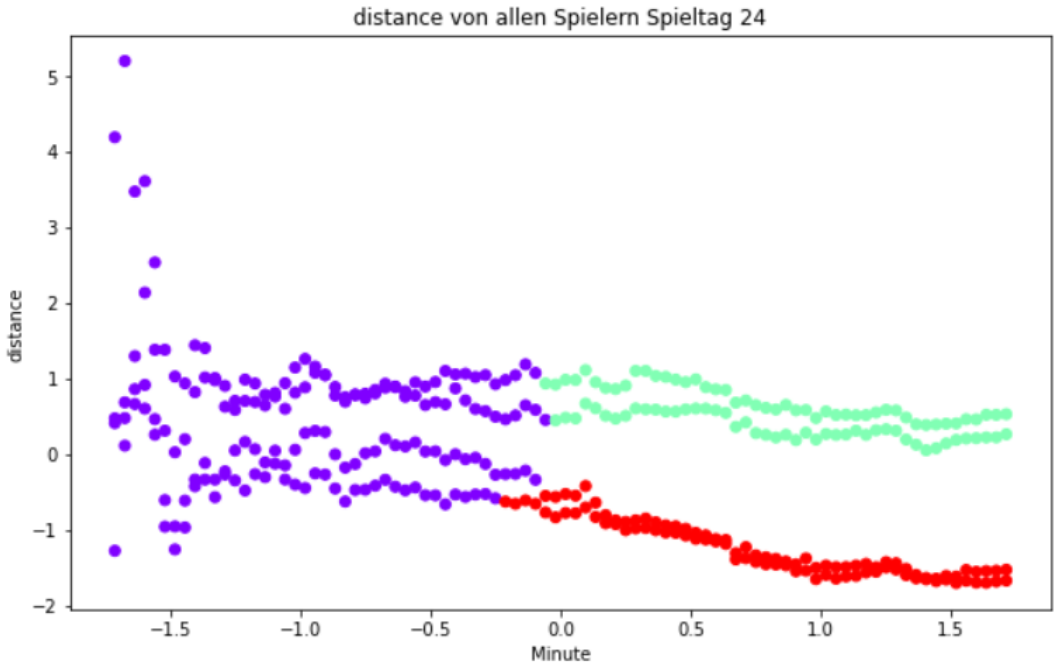

Abbildung 10: KMeans Clustering von *distance* aller Verteidiger an Spieltag 24. (Eigendarstellung)

Doch auch mit dieser Anzahl an Daten war keine Aufteilung der Werte in „Gut“ oder „Schlecht“ sinnvoll möglich. Daher wurde, um mehr Daten zu sammeln, auf den nächsten Spieltag gewartet. Nach dem 25. Spieltag stand ein Dataframe von 720 Zeilen zur Verfügung welcher insgesamt vier Spieler beinhaltete. Dieser Datensatz wurde beispielhaft mit dem Hierarchischen Clustering und DBSCAN geclustert.

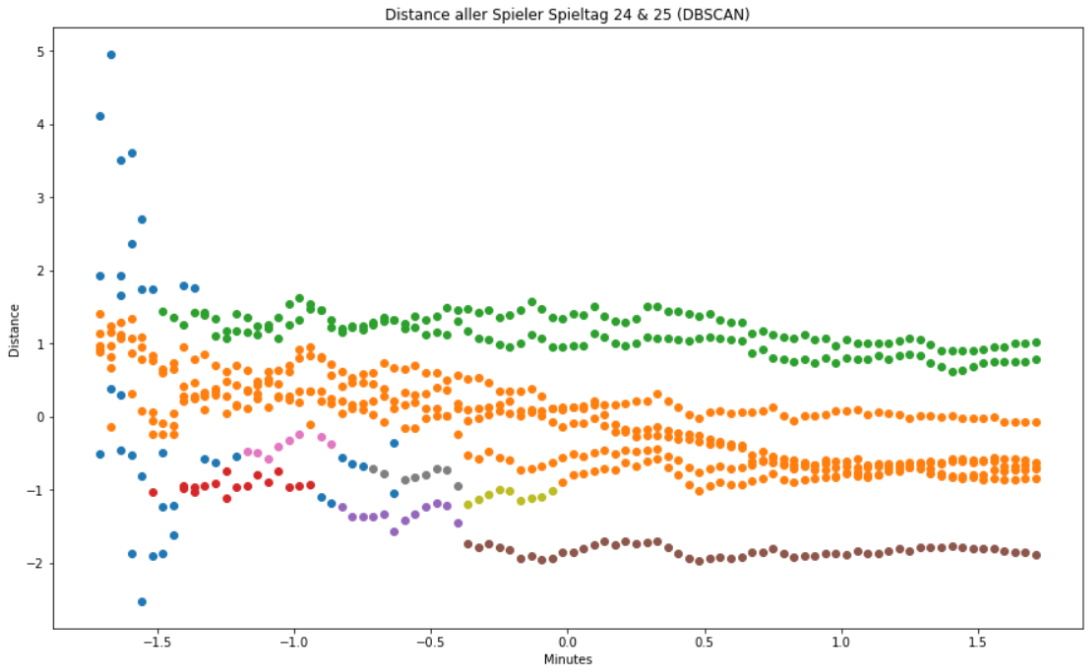

Abbildung 11: DBSCAN Clustering von *distance* aller Verteidiger der Spieltage 24. und 25. (Eigendarstellung)

Die mit DBSCAN gebildeten Cluster waren stark unterschiedlich in ihrer Größe und einige Werte wurden als Ausreißer klassifiziert. Es wäre zwar möglich mit diesem Ergebnis die Bereiche „Gut“, „Schlecht“ und „Durchschnittlich“ zu identifizieren, doch innerhalb dieser Bereiche sind noch Ausreißer zu finden, welche zu falschen Klassifizierungen führen könnten. Es würde bezüglich der Klassifikation keinen Sinn ergeben, einen Wert in der Minute 14 als Ausreißer anzusehen und einen minimal unterschiedlichen Wert in der Minute 15 als „Durchschnittlich“ zu klassifizieren. Dieses Clustering machte klar, dass für ein dichtebasiertes Clustering wie DBSCAN noch zu wenig Daten vorhanden waren und das für dieses Verfahren eine genauere Anpassung der Parameter *eps* und *min_samples* notwendig ist.

Das Hierarchische Clustering schlug mit dem Dendrogramm vier Cluster vor.

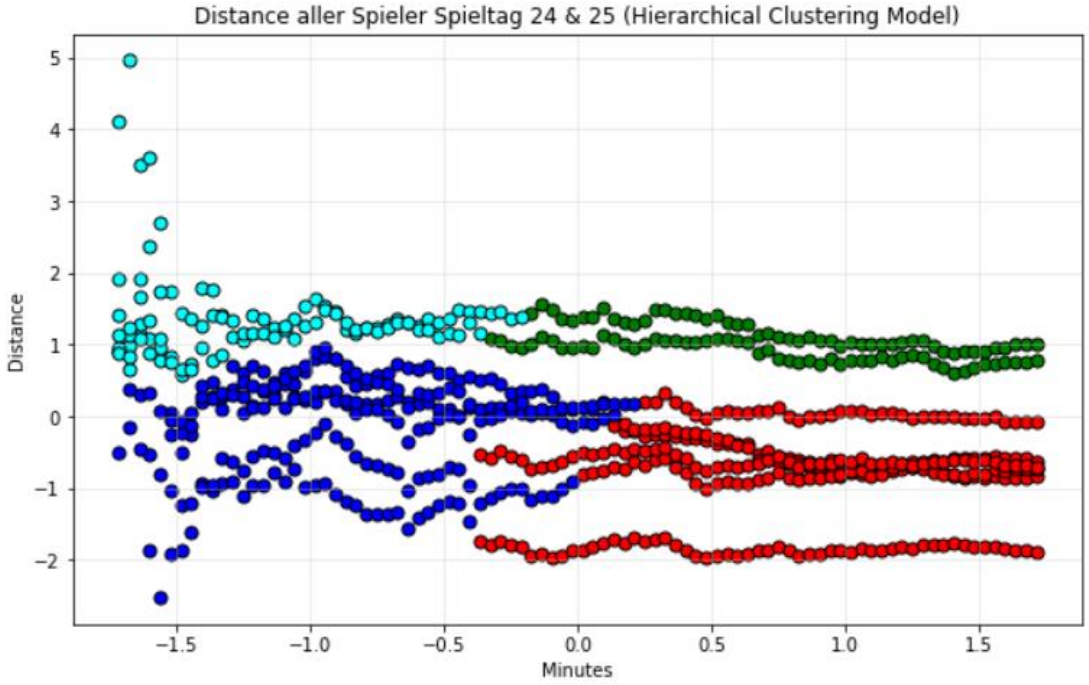

Abbildung 12: Hierarchisches Clustering von *distance* aller Verteidiger der Spieltage 24. und 25. (Eigendarstellung)

Bei diesem Ergebnis war es möglich die unteren beiden Cluster als die „Schlechten“ Werte zu interpretieren und die oberen beiden als die „Guten“ Werte. Doch es stellte sich nun die Frage, ob dies sinnvoll wäre. Denn wie schon in der explorativen Analyse bemerkt wurde, ändert sich die Leistungsfähigkeit eines Spielers im Laufe eines Spiels. So ist es bekannt, dass die Leistung von Spielern bezüglich bestimmter Attribute wie *sprints* mit zunehmender Spielminute abnimmt (Armatas, Yiannakos, & Sileloglou, 2007). Es wäre daher sinnvoll die Werte der Spieler in verschiedenen Minuten-Intervallen zu clustern, um diese Minutenspezifischen Anstiege oder Abstiege zu berücksichtigen. Nach Absprache mit dem HSV kamen wir zu dem Ergebnis, dass eine Unterteilung der vollen 90 Minuten in 15 Minuten-Intervalle am sinnvollsten wäre. So entstehen insgesamt sechs Intervalle (von Minute 0 bis Minute 15, von Minute 16 bis Minute 30, von Minute 31 bis Minute 45 usw.) mit drei Intervallen je Halbzeit, welche dann jeweils die Anfangsphase einer Halbzeit, den Mittelteil einer Halbzeit und das Ende einer Halbzeit repräsentieren.

Um zusätzlich mehr Werte pro Spieler und Spiel in den 15 Minuten-Intervallen zu bekommen, wurde das, unter 2.3.2 beschriebenes Abfrage-Programm, *HZ1min.py* so umgeschrieben, dass es nun anstatt einer Abfrage pro Minute drei machte. Dieses Programm *20Sek_HZ1.py* wurde ebenfalls unter 2.3.2 beschrieben. Das Muster der Abfragen sieht wie folgt aus: Die erste Abfrage wird unter der Minute n.0 abgespeichert, die zweite unter n.2 und die dritte unter n.4. Es erfolgt also alle 20 Sekunden eine Abfrage der Datenbank *dfldata*.

	Zeit	passesLastThird	shotsAtGoal	assitsShotAtGoal	sprints	averageSpeed	maxSpeed	intenseRuns	completedPasses	distance
0	1.0	0	0	0	0.0	7.80	16.45	0.0	0.0	130.85
1	1.2	0	0	0	0.0	7.46	16.45	0.0	0.0	125.36
2	1.4	0	0	0	0.0	7.46	16.45	0.0	0.0	125.36
3	2.0	0	0	0	0.0	7.90	18.42	0.0	0.0	133.73
4	2.2	0	0	0	0.0	8.02	18.42	0.0	0.0	134.41

Abbildung 13: Aufbau der Dataframes mit der Abfrage alle 20 Sekunden (Eigendarstellung)

2.6.1 Vorgegebene Anzahl an Clustern

Um die nachfolgenden Clustering Verfahren vergleichbar zu machen, wurden sie alle auf demselben Datensatz durchgeführt, welcher die Spieltage 24 bis 28 beinhaltet. Dieser enthält insgesamt 3079 Zeilen. Es wurden hierbei auch Auswechslungen in den Datensatz mit einbezogen. Da aus den Daten nicht ersichtlich wird, ob es sich um einen positionsgetreuen Wechsel handelt, werden der Einfachheit halber aller Spieler, die für einen bekannten Verteidiger eingewechselt werden, unabhängig davon, ob sie tatsächlich die Rolle des Verteidigers übernehmen als Verteidiger angesehen. Der Datensatz wurde vor dem Clustering in sechs 15 Minuten-Intervalle aufgeteilt. Diese enthalten minimal 420 Zeilen und maximal 666 Zeilen. Die Unterschiede in den Längen der Datensätze kommen daher zustande, dass die Spieltage 24 und 25 pro Spieler maximal 90 Werte beinhalten, während die folgenden Spieltage 270 Werte beinhalten. Außerdem wurde der Spieltag 27 nicht komplett aufgezeichnet, sondern erst ab der 60. Minute.

KMeans

Angefangen wurde mit dem Clustering des Minuten-Intervalls 0-15. Durch die Elbow-Methode wurde die ideale Anzahl an Clustern von drei bestimmt. Die von KMeans gebildeten Cluster

eigneten sich in dieser Form nicht gut für die Unterteilung in die Bereiche „Gut“, „Schlecht“ und „Durchschnittlich“, denn sie wurden nicht horizontal voneinander getrennt, daher würde die Unterteilung in die drei Bereiche keinen Sinn ergeben. Um dieses Problem zu umgehen, wurde die von der Elbow-Methode errechnete ideale Anzahl an Clustern ignoriert und stattdessen die Anzahl der Cluster schrittweise erhöht. Bei vier Clustern ließ sich eine Unterteilung in „Gut“ und „Schlecht“ machen,

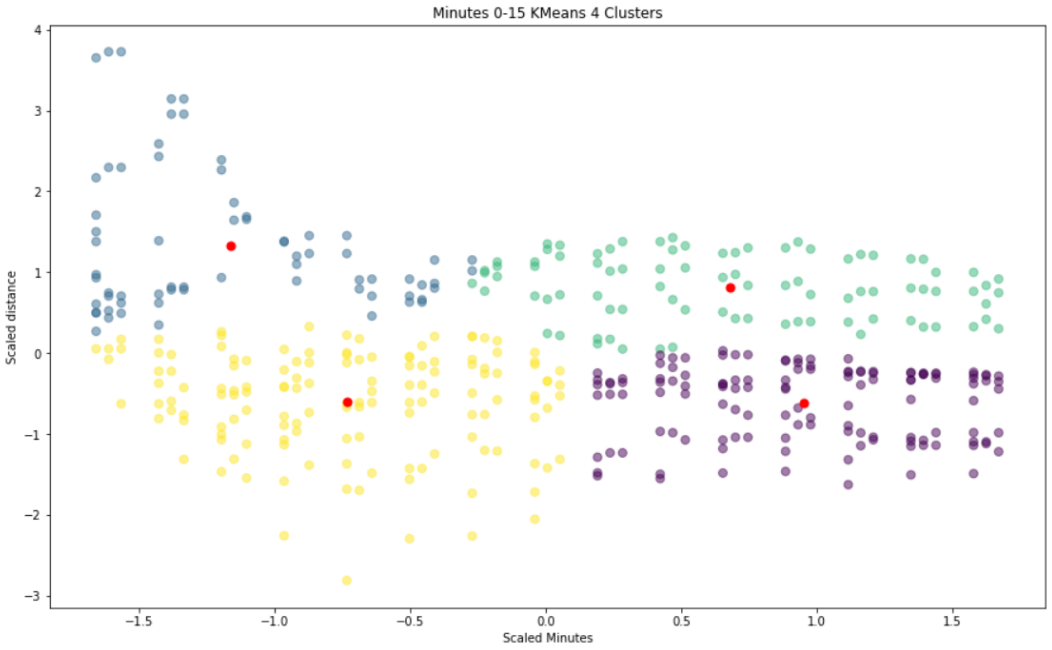

Abbildung 14: KMeans Clustering des Intervalls 0-15 von *distance* mit vier Clustern (Eigendarstellung)

indem die oberen beiden Cluster als „Gute“ Werte angesehen wurden und die unteren beiden als „Schlechte“ Werte. Hierbei müsste dann auf eine Einteilung in „Durchschnittliche“ Werte verzichtet werden. Daher wird die Anzahl der Cluster wieder erhöht. Mit fünf Clustern bilden sich die gewünschten drei horizontalen Bereiche, allerdings sind in den anderen Intervallen wie 15-30 oder auch 30-45 keine sinnvollen Unterteilungen möglich.

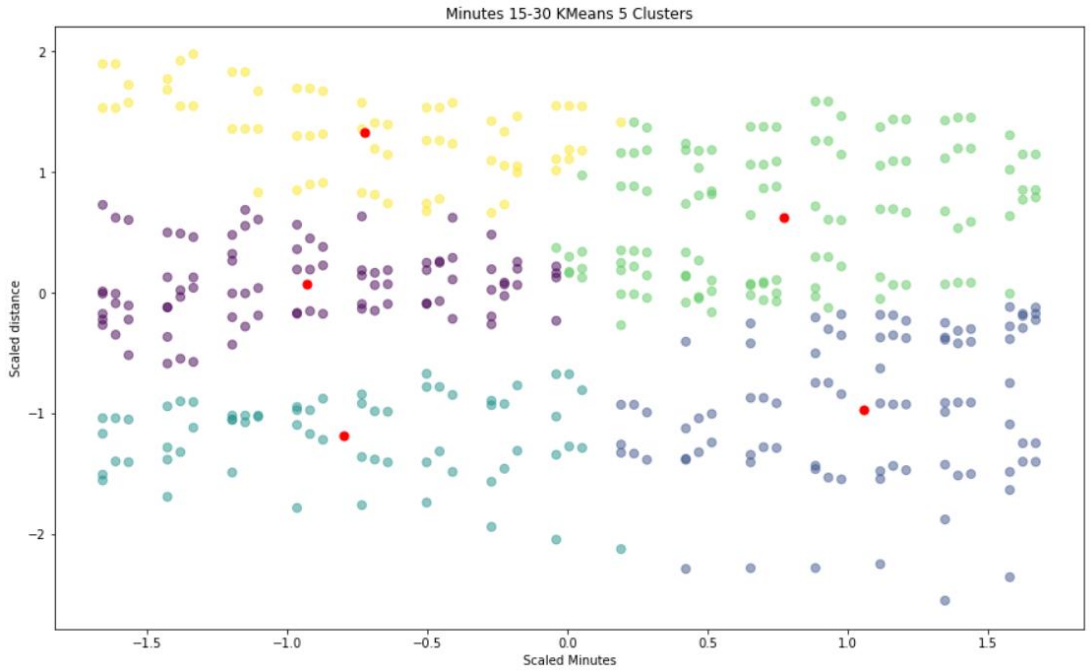

Abbildung 15: KMeans Clustering des Intervalls 15-30 von *distance* mit fünf Clustern (Eigendarstellung)

Hier würden die Bereiche „Gut“ und „Schlecht“ teilweise direkt aneinander grenzen, obwohl diese Werte dem Bereich „Durchschnittlich“ zugeordent werden sollte. Auch in dem Intervall 45-60 führen fünf Cluster zu einer nicht optimalen Unterteilung, denn hier würde der Bereich „Durchschnittlich“ komplett fehlen. Daher wurde die Anzahl der Cluster erneut erhöht. Sechs Cluster eignen sich gut für die Intervalle 15-30 und 30-45, denn hier sind die Werte gleichmäßiger über den gesamten Wertebereich verteilt und KMeans kann die Cluster besser bilden.

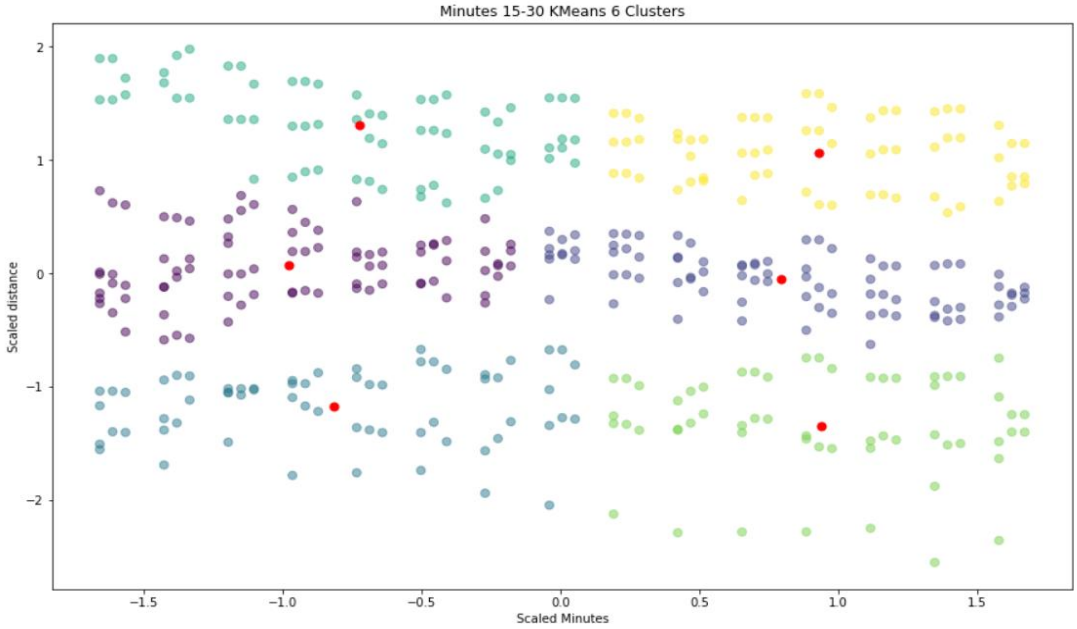

Abbildung 16: KMeans Clustern des Intervalls 15-30 von *distance* mit sechs Clustern (Eigendarstellung)

Bei den Intervallen 45-60 und 60-75 fällt das Ergebnis jedoch nicht gut aus. Hier ist nur eine Unterteilung in die Bereiche „Gut“ und „Schlecht“ möglich.

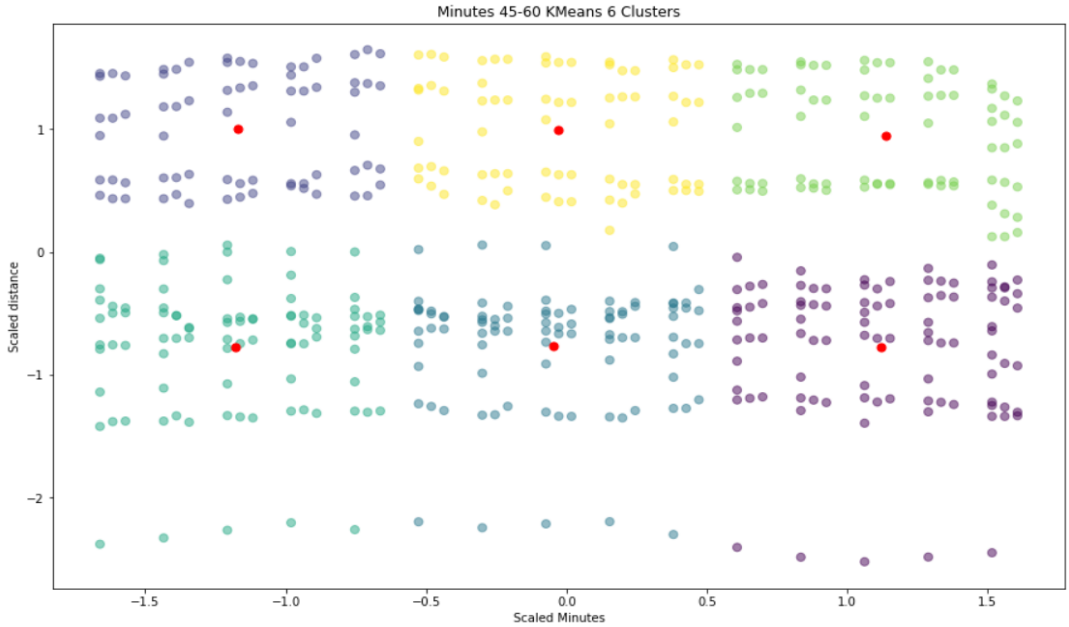

Abbildung 17: KMeans Clustern des Intervalls 45-60 von *distance* mit sechs Clustern (Eigendarstellung

Das größte Problem stellen allerdings die Werte des Minuten-Intervalls 75-90 dar, hier sind fast alle Werte um den X-Wert null gelegen. Logisch wäre hierbei eine die Unterteilung der drei unteren Cluster in die „Schlechten“ Werte, der zwei darüberliegenden Cluster in die „Durchschnittlichen“ Werte und dem oberen Cluster in die „Guten“ Werte.

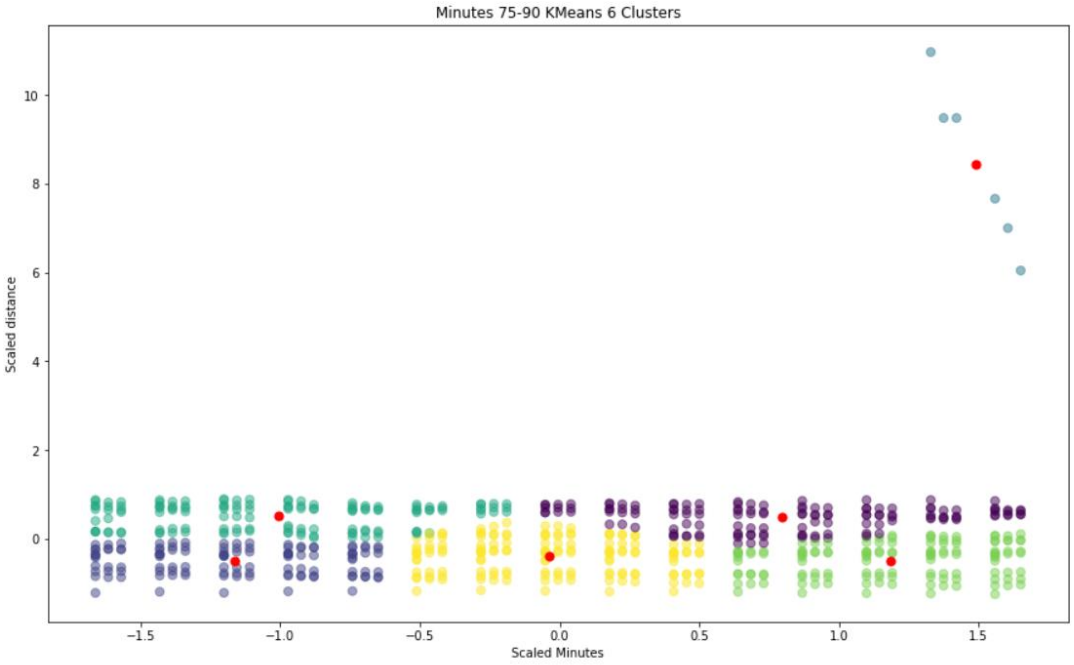

Abbildung 18: KMeans Clustern des Intervalls 75-90 von *distance* mit sechs Clustern (Eigendarstellung)

Da keine Ausreißer Behandlung vorgenommen wurde und auch keine geplant ist fällt das Ergebnis in diesem Intervall schlechter aus. Allerdings sollte, aufgrund des starken Einflusses dieser anscheinenden Ausreißer auf das Ergebnis, erneut eine Ausreißer Behandlung in Erwägung gezogen werden.

Nach einer erneuten Erhöhung der Anzahl der Cluster auf sieben bilden sich auch in dem Intervall 45-60 die drei Bereiche ab.

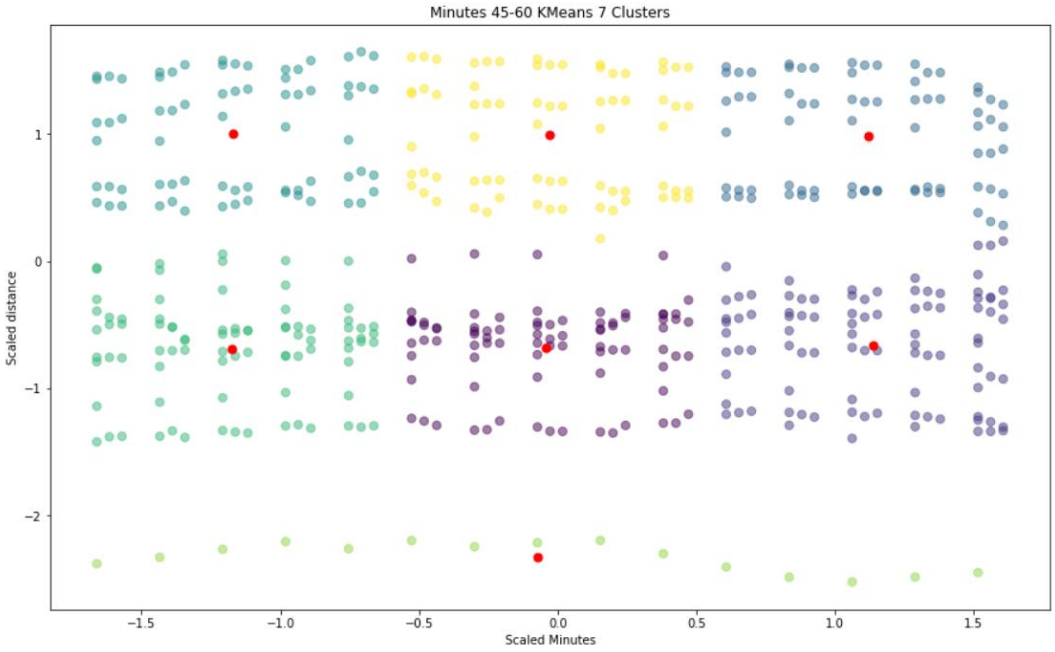

Abbildung 19: KMeans Clustern des Intervalls 45-60 von *distance* mit sieben Clustern (Eigendarstellung)

Allgemein lassen sich mit sieben Clustern in allen Intervallen sinnvollere Unterteilungen in die drei gewünschten Bereiche erkennen. Die einzige Ausnahme bildet das Intervall 75-90. Dieses Intervall wird in den folgenden Verfahren weiter beobachtet, um zum Entschluss zu kommen, ob eine Ausreißer Behandlung sinnvoll ist oder nicht.

Hierarchisches Clustering

Bei dem Hierarchischen Clustering wurde anhand des Dendrogramms eine ideale Anzahl von vier Clustern ermittelt. Das Clustering fiel ähnlich aus, wie das Clustern mit KMeans und vier Clustern. Dementsprechend fehlt auch hier der Bereich der „Durchschnittlichen“ Werte.

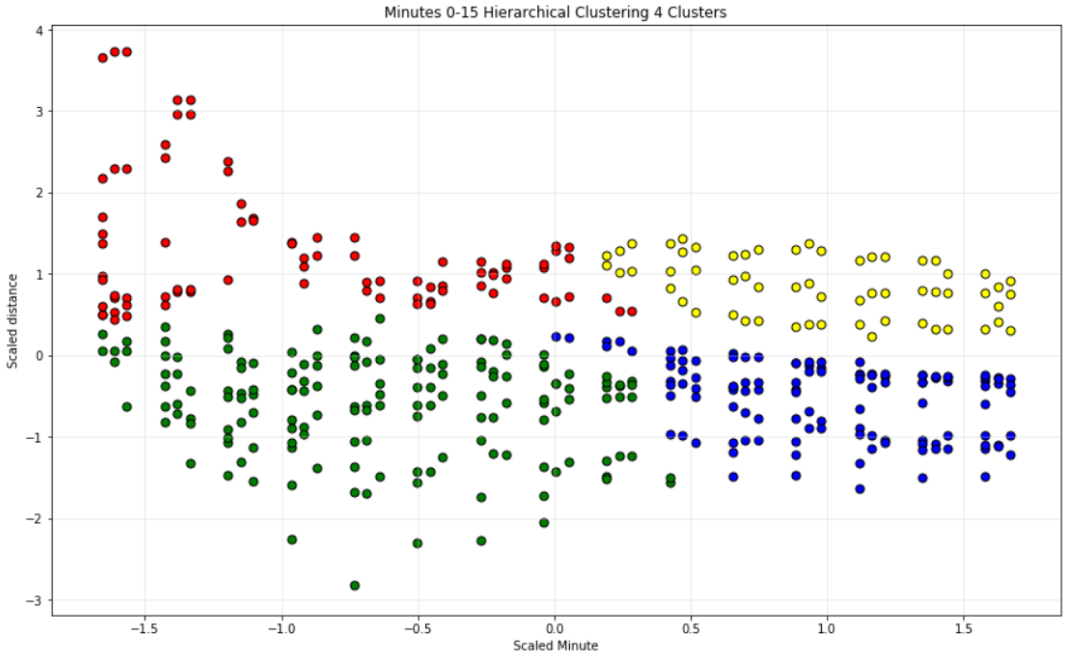

Abbildung 20: Hierarchisches Clustering des Intervalls 0-15 von *distance* mit vier Clustern (Eigendarstellung)

Daher wird auch hier die Anzahl der Cluster erhöht. Bei fünf Clustern ergab sich ebenfalls ein ähnliches Ergebnis zur KMeans Clusterung mit fünf Clustern und auch bei sechs ergaben sich nur minimale Unterschiede zu KMeans. Die Minuten-Intervalle 15-30 und 30-45 fallen bei sechs Clustern jedoch ein wenig schlechter aus als bei KMeans.

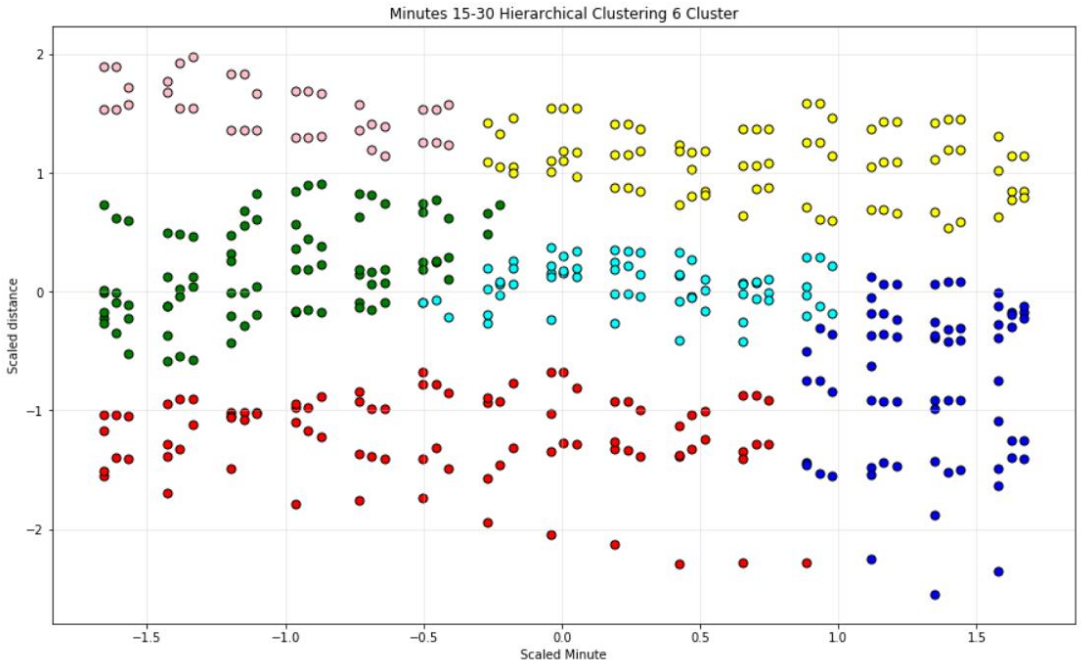

Abbildung 21: Hierarchisches Clustering des Intervalls 15-30 von *distance* mit sechs Clustern (Eigendarstellung)

KMeans hat hier dem Hierarchischen Clustering voraus, dass es seine Cluster eher sphärisch bildet und diese damit gleichmäßiger ausfallen. Beim Hierarchischen Clustering würde hier das blaue Cluster sowohl den Bereich „Schlecht“ als auch den Bereich „Durchschnittlich“ überbrücken. Dadurch würde es zu falschen Klassifikationen kommen, je nachdem, welchem von diesen zwei Bereichen das blaue Cluster tatsächlich zugeordnet werden würde.

Auch mit sieben Clustern fallen die Ergebnisse etwas schlechter aus als bei KMeans.

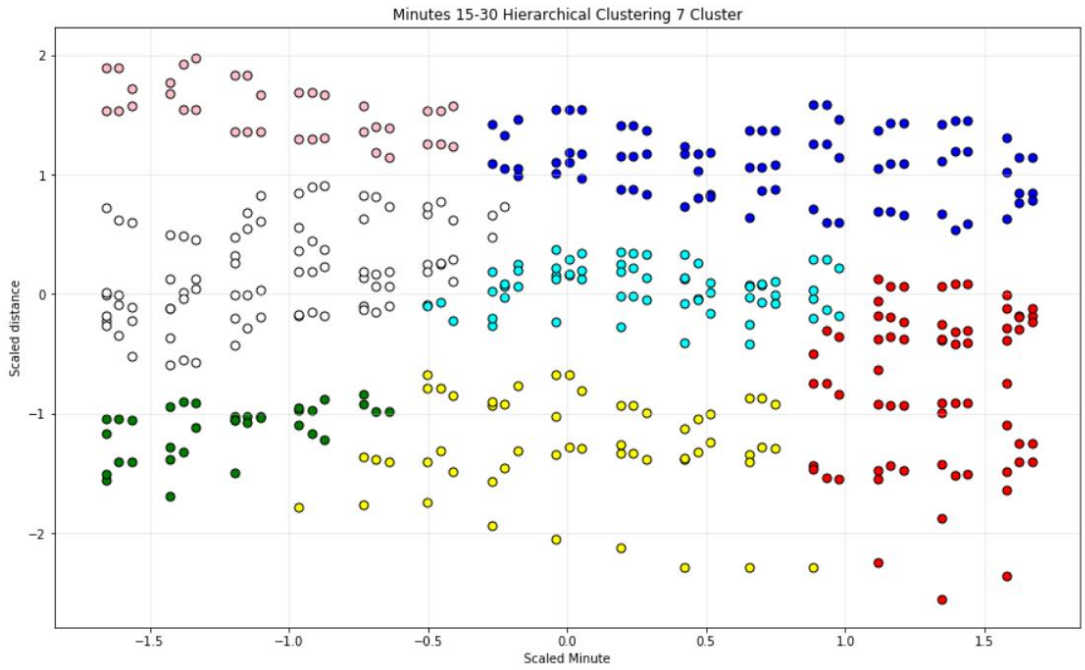

Abbildung 22: Hierarchisches Clustering des Intervalls 15-30 von *distance* mit sieben Clustern (Eigendarstellung)

Da die Cluster nicht so sphärisch gebildet werden kommt es dazu, dass die Bereiche ineinander übergehen. So überbrückt im Intervall 15-30 das rote Cluster sowohl den Bereich „Schlecht“ als auch den Bereich „Durchschnittlich“. Dies kann erneut zu falschen Klassifikationen führen, falls ein Klassifikator auf diesem Clustering aufgebaut werden sollte.

Ebenso wie KMeans erzielt das Hierarchische Clustering bei den Werten des Minuten-Intervalls 75-90 kein gutes Ergebnis. Insgesamt sind die gelieferten Ergebnisse etwas schlechter als jene von KMeans.

BIRCH

Da die vorherigen Verfahren bei vier und fünf Clustern keine ausreichend guten Ergebnisse erzielen konnten, wurde hier nicht mit dieser Anzahl an Clustern geclustert. Stattdessen wurde mit sechs Clustern und Parametern *threshold* = 0.35 und *branching-factor* = 5 geclustert. Das Ergebnis fiel vergleichbar mit den vorherigen beiden Clustering-Verfahren aus. Hier würden

das lila und das grüne Cluster die „Guten“

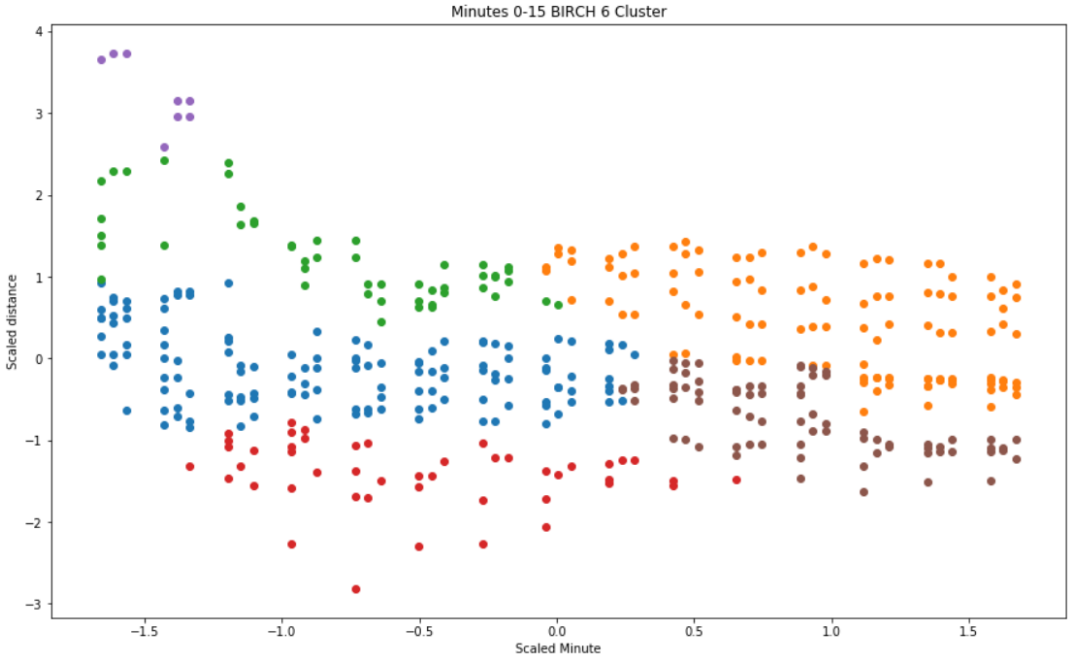

Abbildung 23: BIRCH Clustering des Intervalls 0-15 von *distance* mit sechs Clustern, *branching-factor = 5*, *threshold = 0.35* (Eigendarstellung)

Werte bilden, das rote und das braune die „Schlechten“ und das blaue die „Durchschnittlichen“. Das orange Cluster jedoch könnte zwei Bereichen „Gut“ oder „Durchschnittlich“ zugeordnet werden. Dieses Ergebnis kann mit der Anpassung der Parameter *threshold* auf 0.2 und *branching-factor* auf 10 verbessert werden.

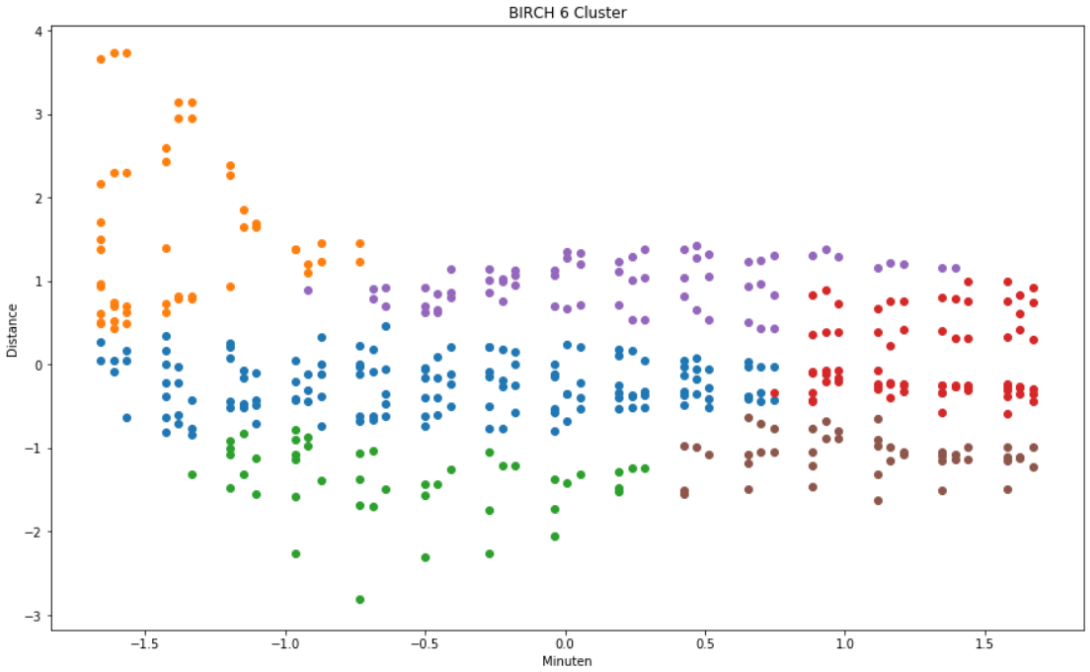

Abbildung 24: BIRCH Clustering des Intervalls 0-15 von *distance* mit sechs Clustern, *branching-factor = 10*, *threshold = 0.2* (Eigendarstellung)

Denn nun würden das orange und das lilane Cluster die „Guten“ Werte bilden, das blaue und das rote die „Durchschnittlichen“ und das grüne und das braune die „Schlechten“. Das rote Cluster beinhaltet weiniger stark die zwei Bereiche „Gut“ und „Durchscnittlich“ als das orange Cluster es im vorherigen Durchlauf tat, dementsprechent fällt das Ergebnis ein wenig besser aus. Mit BIRCH lassens sich durch Anpassungen der Parameter bei fast allen Minuten-Intervallen gute Ergebnisse erzielen. Bei dem Intervall 15-30 ist es *threshold* = 0.1 und *branching-factor* = 10, bei dem Intervall 30-45: *threshold* = 0.2 und *branching-factor* = 5, bei dem Intervall 45-60: *threshold* = 0.5 und *branching-factor* = 6.

Auch mit sieben Clustern sind für fast jedes Intervall gute Ergebnisse zu erzielen, allerdings bedarfen diese auch hier Anpassungen der Parameter. Für die Intervalle 0-15 und 60-75 benötigt es einen *thershold* von 0.4 und einen *branching-factor* von 8, für die Intervalle 15-30, 30-45 und 45-60 benötigt es einen *thershold* von 0.5 und einen *branching-factor* von 6, lediglich für das Intervall 75-90 lassen sich nur maximal akzeptable Ergebnisse finden.

Insgesamt ist es mit BIRCH möglich, was die beobachteten Intervalle und Attribute angeht, in etwa ähnlich gute Ergebnisse wie bei KMeans und zu erzielen. Diese guten Ergebnisse bedarfen allerdings ständiger Anpassung der Parameter *threshold* und *branching-factor*. Die ständige Anpassung dieser ist jedoch bei dem geplanten Klassifikator keine Option.

Spectral Clustering

Das Intervall 0-15 fällt beim Spectral Clustering mit sechs Clustern ähnlich aus wie bei den vorherigen Verfahren. Es werden zwei Cluster gebildet die als „Schlecht“ interpretiert werden können, zwei die als „Durchschnittlich“ interpretiert werden können und zwei die als „Gut“ interpretiert werden können. Daher werden die nachfolgenden Intervalle betrachtet. Die Intervalle 15-30 und 30-45 werden annehmbar geclustert, sie lassen die Unterteilung in die gewünschten Bereiche zu, jedoch unterscheiden sich die Cluster in der Größe stärker als bei den bisher beobachteten Verfahren. Auch kommt es vermehrt zu Überläufen der Cluster in andere Bereiche.

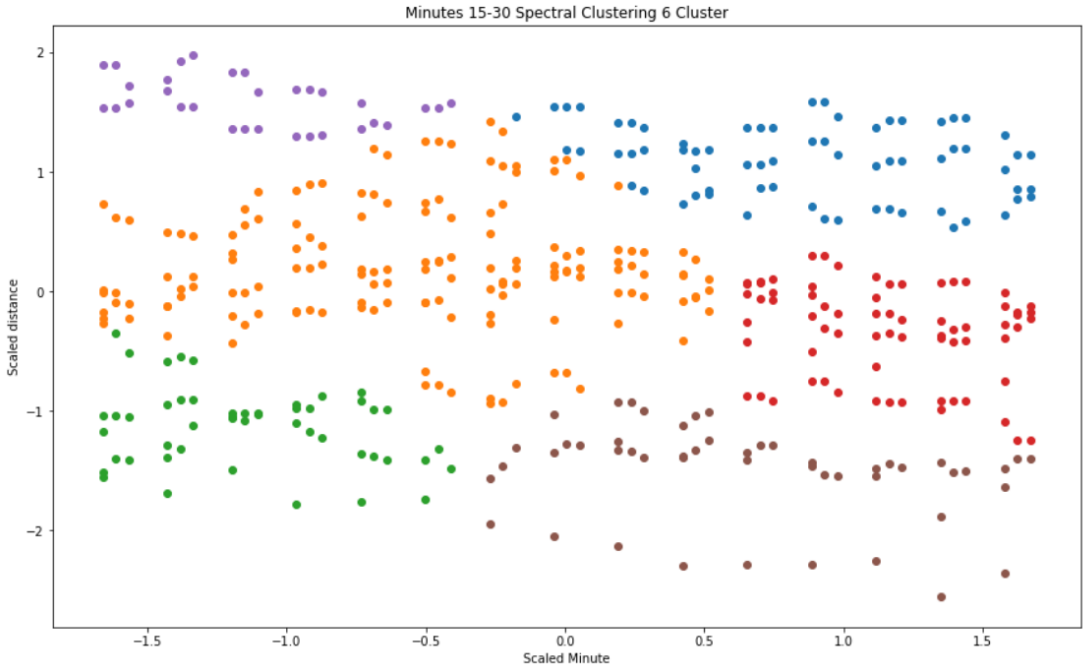

Abbildung 25: Spectral Clustering des Intervalls 15-30 von *distance* mit sechs Clustern (Eigendarstellung)

Auch bei den Intervallen 45-60 und 60-75 ist eine logische Unterteilung in die gewünschten Bereiche möglich. Ebenfalls lässt sich bei diesen beobachten, dass die Werte je eines Clusters, welches dem Bereich „Durchschnittlich“ zugeordnet werden würde, in dem Wertebereich des Bereiches „Gut“ liegen. Daher könnte es dazu kommen, dass bei dem Klassifikator derselbe Wert einmal als „Gut“ und ein anderes Mal als „Durchschnittlich“ klassifiziert werden würde.

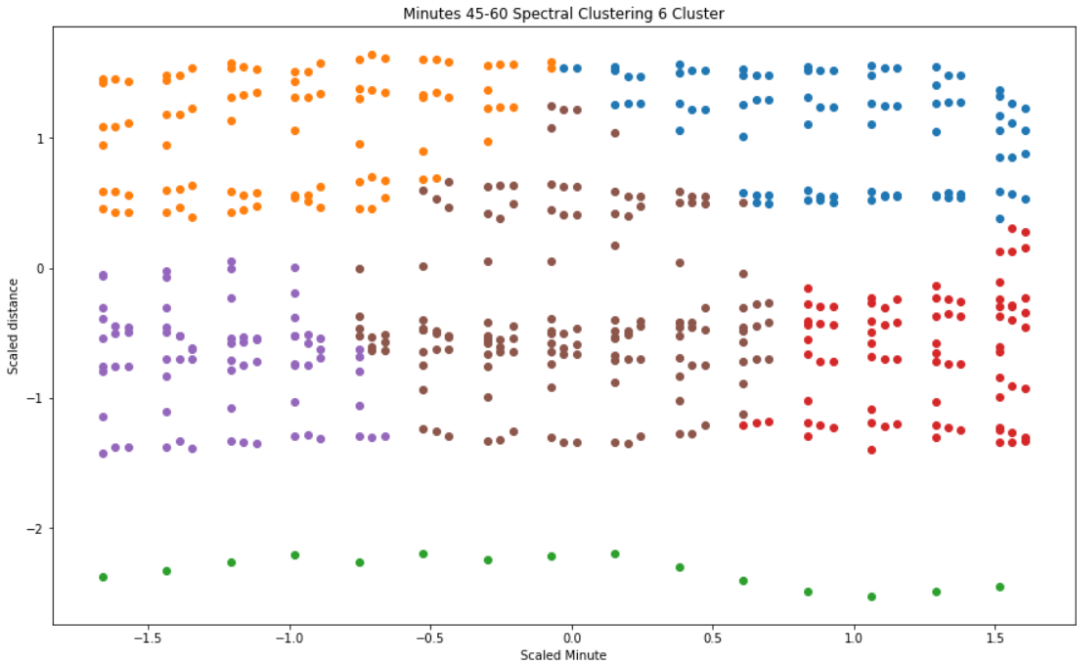

Abbildung 26: Spectral Clustering des Intervalls 45-60 von *distance* mit sechs Clustern (Eigendarstellung)

Bei dem Intervall 75-90 ist auch beim Spectral Clustering keine logische Unterteilung in die Bereiche „Gut“, „Schlecht“ und „Durchschnittlich“ möglich.

Im Gegensatz zu den anderen Verfahren werden die Ergebnisse mit sieben Clustern beim Spectral Clustering deutlich schlechter. Hier bilden sich in den Intervallen 15-30, 30-45 und 60-75 große Bereichsübergreifende Cluster, die nicht logisch interpretierbar sind.

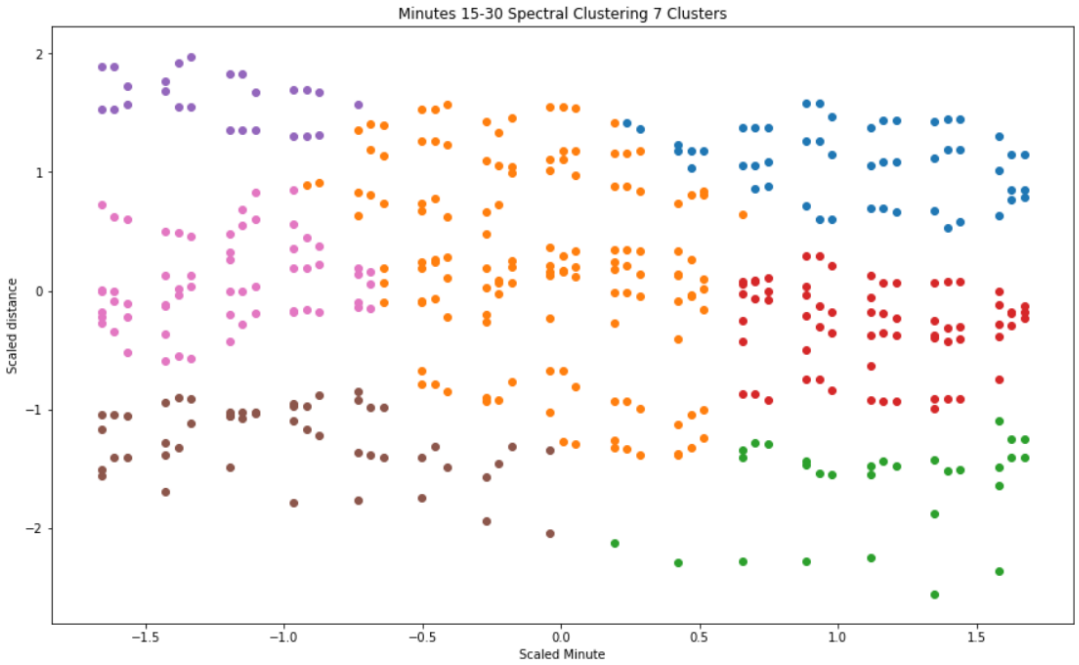

Abbildung 27: Spectral Clustering des Intervalls 15-30 von *distance* mit sieben Clustern (Eigendarstellung)

Wie in dem Intervall 15-30, hier beinhaltet das orange Cluster offensichtlich mehrere Bereiche und kann somit nicht logisch nur einem Bereich zugeordnet werden.

Allgemein sind die vom Spectral Clustering gebildeten Cluster nicht so gut wie die der vorherigen Verfahren.

2.6.2 Selbs bestimmende Anzahl an Clustern

DBSCAN

Das Clustering des Minuten-Intervalls 0-15 wurde mit den Parametern *eps* = 0.25 und *min_samples* = 5 durchgeführt. Hier lassen sich die gewünschten Bereiche zwar erkennen,

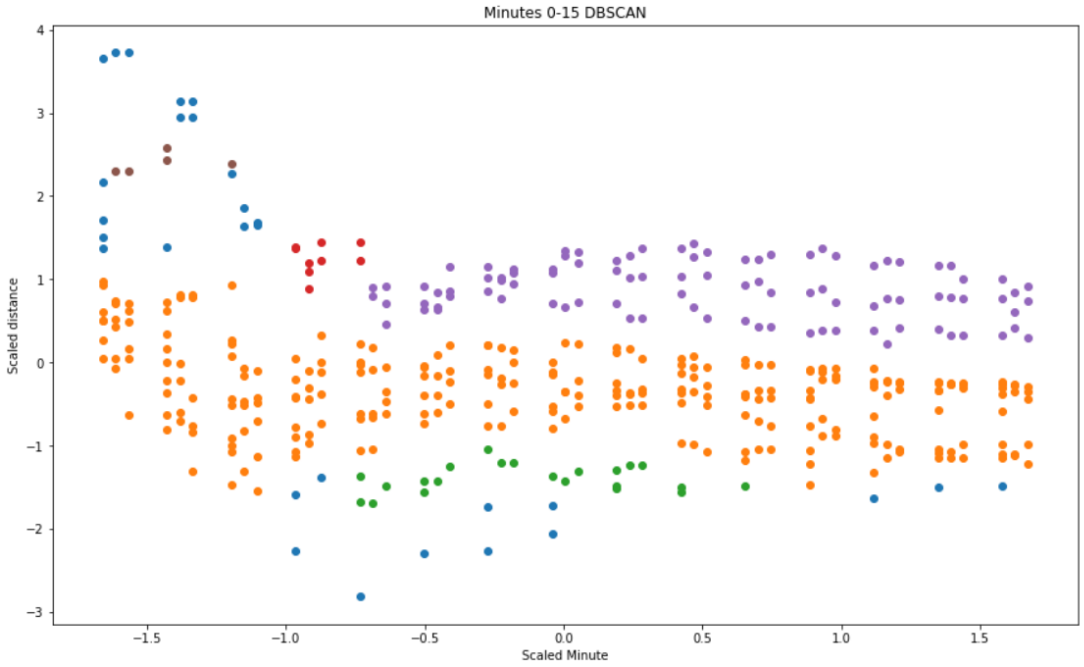

Abbildung 28: DBSCAN Clustering des Intervalls 0-15 von *distance*, *eps* = *0.25*, *min_samples* = *5* (Eigendarstellung)

doch es sind auch viele Ausreißer vorhanden (in blau). Die zwei kleineren Cluster (rot und braun) und das lila Cluster würden in diesem Fall dem Bereich „Gut“ zugeordnet werden, das orange dem Bereich „Durchschnittlich“ und das grüne dem Bereich „Schlecht“. Die Ausreißer würden keinem Bereich zugeordnet werden. Es könnte ein Problem darstellen, dass einige Werte, die keine Ausreißer sind, von DBSCAN als solche angesehen werden. Dadurch könnte es bei der späteren Klassifizierung zu Fehlern kommen. Das Intervall 30-45 wird als einziges Intervall besser geclustert, hier kommt es zu drei horizontal getrennten Clustern, welche den drei Bereichen zugeordnet werden können, doch auch hier sind Ausreißer vorhanden. In den Intervallen 15-30, 45-60 und 75-90 kommt es mit denselben Parametern zu schlechten Ergebnissen, in welchen nur ein Cluster oder zwei Cluster erkannt werden.

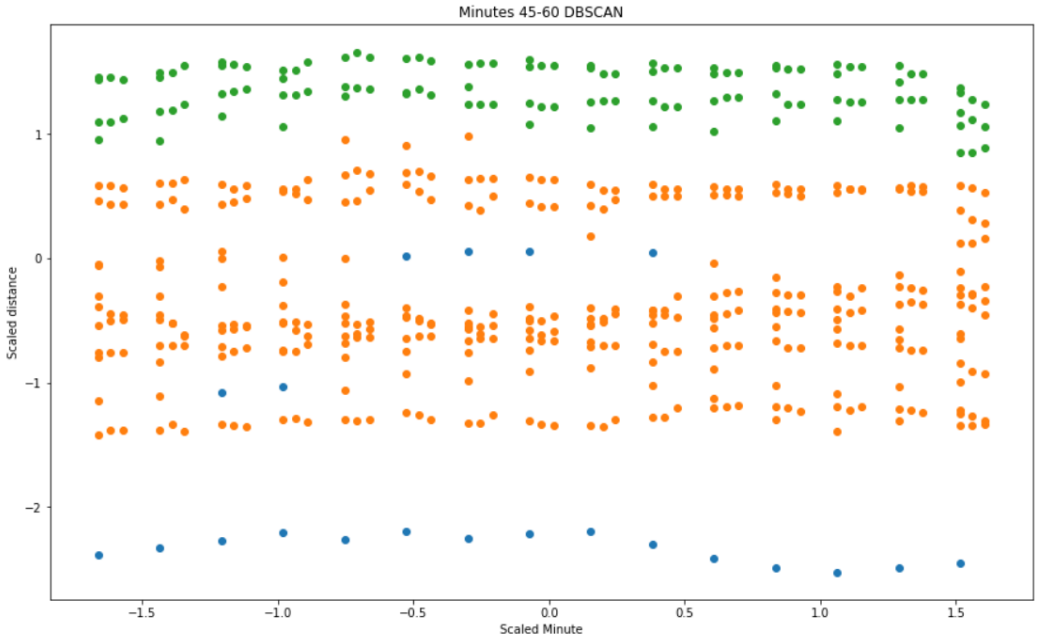

Abbildung 29: DBSCAN Clustering des Intervalls 45-60 von *distance*, *eps = 0.25*, *min_samples = 5* (Eigendarstellung)

So wäre im Intervall 45-60 nur eine Aufteilung in die Bereiche „Gut“ und „Schlecht“ möglich, wobei erneut Ausreißer vorhanden wären, bei welchen es sich logisch gesehen um keine handelt.

Für die restlichen Intervalle lassen sich durch teils stärkere Anpassungen der Parameter *eps* und *min_samples* angemessene Ergebnisse erzielen. Für das Intervall 15-30 ist hierfür ein *eps* von 0.24 und *min_samples* von 5 nötig. Es werden vier Cluster gebildet von welchen die unteren beiden als Bereich „Schlecht“, das mittlere als „Durchschnittlich“ und das obere als „Gut“ interpretiert werden können.

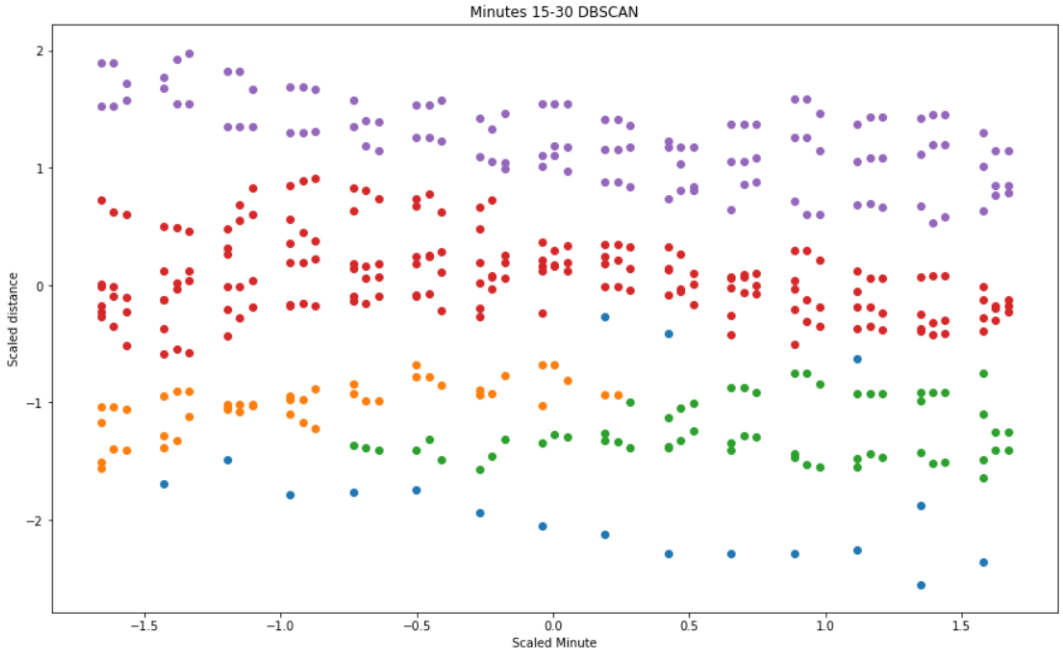

Abbildung 30: DBSCAN Clustering des Intervalls 15-30 von *distance*, *eps = 0.24*, *min_samples = 5* (Eigendarstellung)

Für das Intervall 45-60 bedarf es einen *eps* Wert von 0.24 und ein *min_samples* von 6, bei dem Intervall 60-75 beträgt *eps* für akzeptable Ergebnisse 0.15 und *min_samples* 6 und bei dem Intervall 75-90 liegt *eps* bei 0.16 und *min_samples* bei 5. In allen diesen Ergebnissen tauchen weiterhin Ausreißer auf. Bei Ausreißern die nahe an den gebildeten Clustern liegen ist dies damit zu erklären ist, dass noch nicht genug Daten vorhanden sind, um mit DBSCAN und seinem dichtebasierten Vorgehen diese Werte den Clustern zuzuordnen. Bei Ausreißern, die weiter von den gebildeten Clustern entfernt liegen könnte es sich tatsächlich um Ausreißer handeln, die nicht behandelt werden. Es ist zu erwarten, dass mit einer größeren Menge an Daten die Ergebnisse verbessert werden können, doch der Bedarf der Anpassung der Parameter würde wohl auch mit mehr Daten nicht entfallen. Hierbei müsste nicht nur die Änderungen von Intervall zu Intervall eines Attributes bedacht werden, sondern vor allem die Änderungen, die für die Verschiedenen Attribute notwendig sind. Denn damit DBSCAN gute Ergebnisse liefert muss besonders der Parameter *eps* dem Wertebereich des jeweiligen Attributes entsprechend angepasst werden.

Affinity Propagation

In dem Intervall 0-15 wurde das Clustering mit den Parametern *damping* = 0.6 und *preference* = -50 durchgeführt. Das Ergebnis fiel ähnlich jenem mit KMeans und fünf Clustern aus.

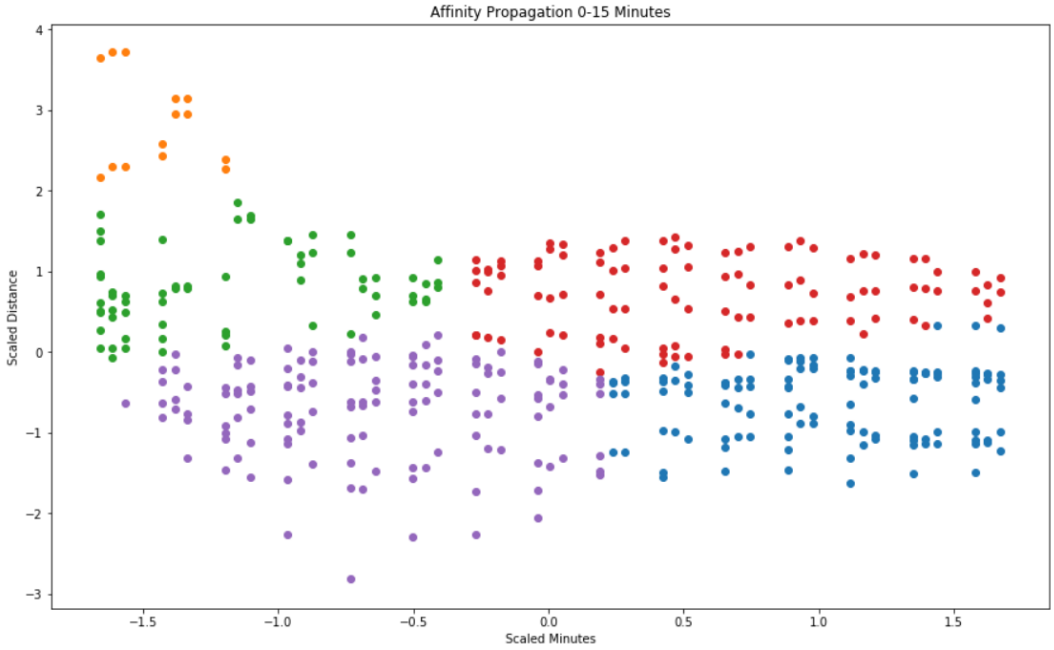

Abbildung 31: Affinity Propagation Clustering des Intervalls 0-15 von *distance*, *damping = 0.6*, *preference = -50* (Eigendarstellung)

Hierbei könnten die zwei unteren Cluster dem Bereich „Schlecht“ zugeordnet werden, die zwei mittleren dem Bereich „Durchschnittlich“ und das obere dem Bereich „Gut“. Die restlichen Intervalle fallen für diese Parameter jedoch schlechter aus. Bei den Intervallen 15-30 und 45-60 ist nur eine Unterteilung in die Bereiche „Gut“ und „Schlecht“ möglich und die übrigen Intervalle lassen aufgrund von bereichsübergreifenden Clustern nur wenig sinnvolle Unterteilung zu. Damit in den Intervallen ab Minute 15 bessere Ergebnisse erzielt werden müssen die Parameter *damping* und *preference* angepasst werden.

Mit einem *damping* von 0.8 und einer *preference* von -50 lässt sich für das Intervall 15-30 ein gutes Ergebnis erzielen, wobei sich sechs Cluster bilden, die die Bereiche horizontal voneinander trennen. Für ein gutes Ergebnis bedarf es im Intervall 30-45 ein *damping* von 0.5 und eine *preference* von -30. Bei dem Intervall 45-60 ist das beste Ergebnis mit einem *damping* von 0.5 und einer *preference* von -6 erreicht, wobei auch hier noch Cluster die Bereichsgrenzen

übergreifen, wie das orange Cluster.

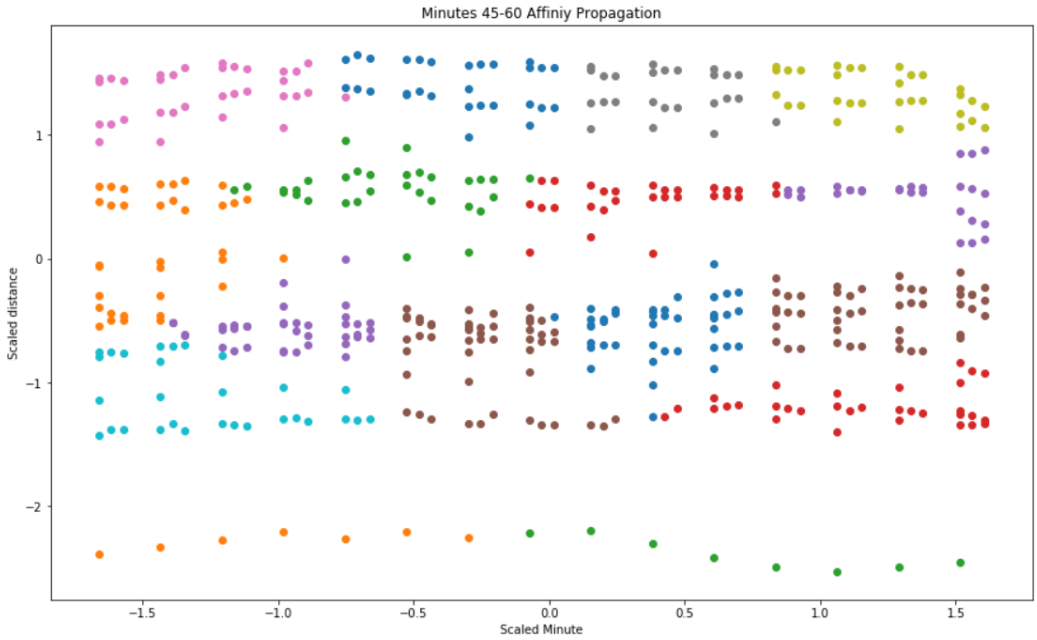

Abbildung 32: Affinity Propagation Clustering des Intervalls 45-60 von *distance*, *damping = 0.5*, *preference = -6* (Eigendarstellung)

Auch für das Intervall 60-75 lassen sich durch Anpassungen der Parameter *damping* und *preference* angemessene Ergebnisse erzielen, bei diesem beträgt das *damping* = 0.7 und die *preference* = -30.

Es kommt also auch hier, wie bei DBSCAN das Problem auf, dass logische Ergebnisse von Intervall zu Intervall Anpassungen der Parameter benötigen und mit gleichbleibenden Parametern keine sinnvollen Ergebnisse in allen Intervallen erreicht werden können. Dementsprechend ist zu erwarten, dass gleichbleibende Parameter auch Attributs-übergreifend nicht die gewünschten Ergebnisse liefern würden.

MeanShift

MeanShift liefert bei dem Intervall 0-15 mit einem *quantile* Wert von 0.15 ein ähnliches Ergebnis wie KMeans mit vier Clustern. Eine logische Aufteilung in die gewünschten Bereiche wäre bei diesem Intervall dementsprechend nur in zwei Bereiche möglich. Dieser *quantile* Wert liefert allerdings für die anderen Intervalle keine optimalen Ergebnisse.

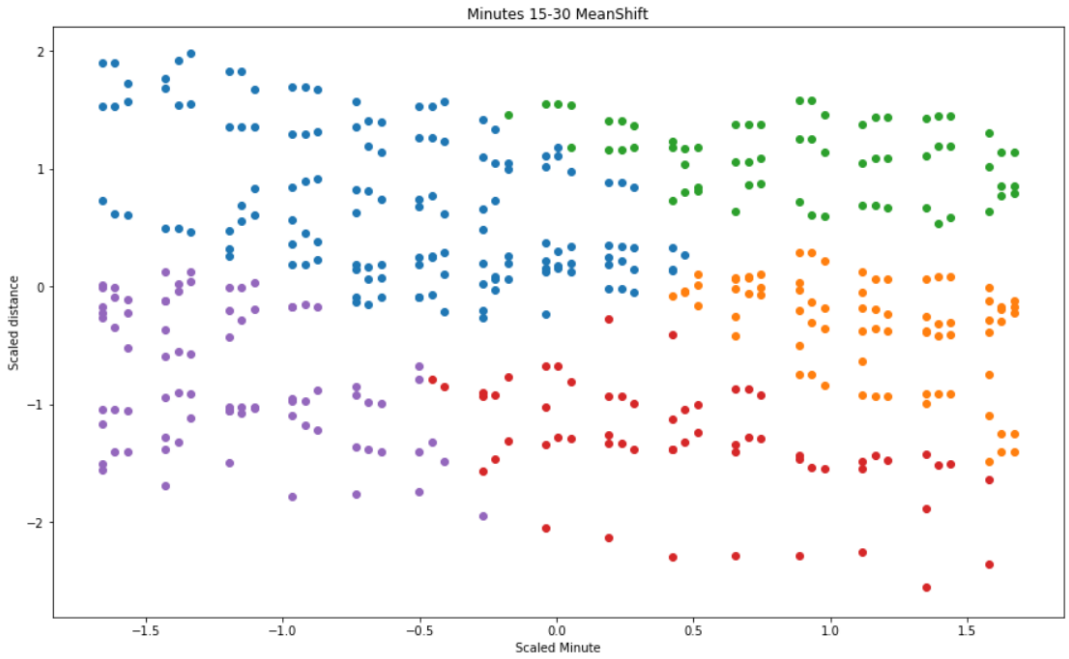

Abbildung 33: MeanShift Clustering des Intervalls 15-30 von *distance, quantile = 0.15* (Eigendarstellung)

So beinhalten in dem Intervall 15-30 die Werte des blauen Clusters aus dem Bereich „Gut“ einige Werte, die logisch als „Durchschnittlich“ eingestuft werden sollten. Ebenso würde das lila Cluster des Bereiches „Schlecht“ einige dieser „Durchschnittlichen“ Werte beinhalten. Somit würden nur Werte als „Durchschnittlich“ angesehen werden, wenn sie dem orangen Cluster entsprechen.

Bei Anpassung des Parameters *quantile* auf 0.1 werden die Cluster horizontaler gebildet als vorher, damit wird der Bereich „Durchschnittlich“ besser abgebildet.

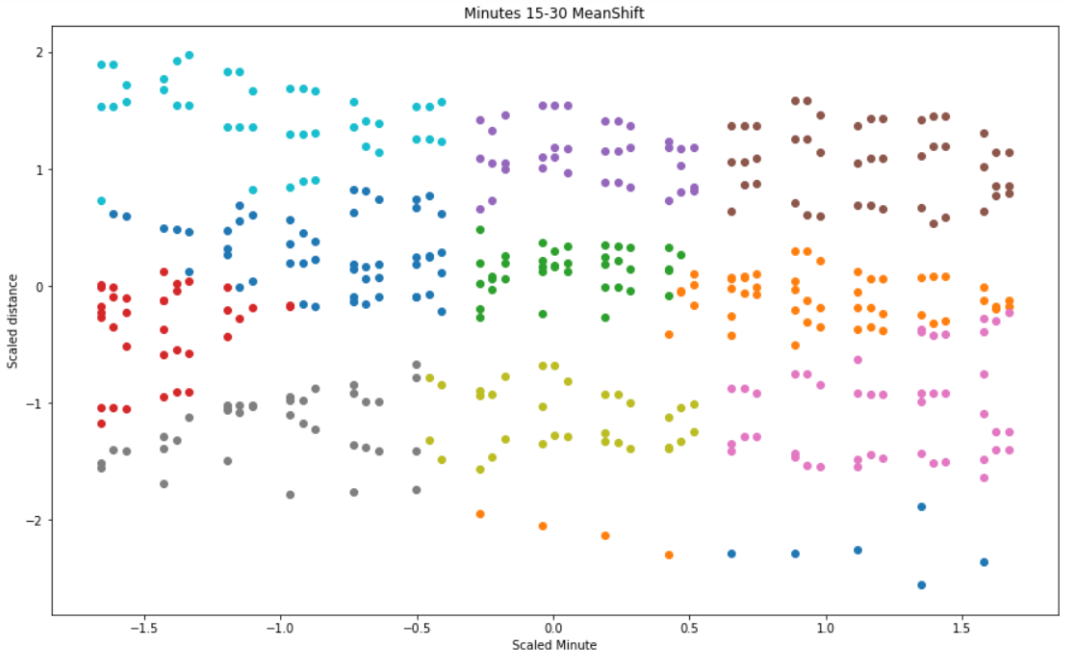

Abbildung 34: MeanShift Clustering des Intervalls 15-30 von *distance*, *quantile = 0.1* (Eigendarstellung)

Durch diese Anpassung des Parameters *quantile* auf 0.1 werden in allen Intervallen mehr kleinere Cluster gebildet als vorher, welche besser in die Bereiche „Gut“, „Schlecht“ und „Durchschnittlich“ unterteilt werden können. Es werden mit diesem Parameter in allen Intervallen, bis auf das Intervall 75-90, angemessene bis gute Ergebnisse erzielt.

Allgemein fallen die Ergebnisse mit MeanShift daher akzeptabel bis gut aus. Die höhere Anzahl an gebildeten Clustern mit dem *quantile* 0.1 führt jedoch dazu, dass das Clustern länger dauert als bei anderen Verfahren mit ähnlich guten Ergebnissen.

2.7 Diskussion

Bei allen Verfahren schnitt das Intervall 75-90 nicht gut ab, die hier anscheinend aufkommenden Ausreißer sind Werte eingewechselten Spieler, welche im Vergleich zu Spielern der Startelf natürlich mehr Energie in diesen späten Minuten aufbringen können. Dieses Intervall wurde

daher bei den Verfahren nicht ausschlaggebend berücksichtigt. Bei dem Bau des Klassifikators werden dennoch diese eingewechselten Spieler nicht entfernt, da für jeden Spieler sein persönlicher Dataframe erstellt wird, auf welchem dann geclustert wird, und diese Werte ebenso wie die anderen zu den Werten dieses Spielers gehören.

Für die Klassifikatoren entfallen die Verfahren, welche für gute oder akzeptable Ergebnisse eine Anpassung von Parametern von Intervall zu Intervall benötigen, denn dies kann nicht gewährleistet werden. Daher bleiben noch die Verfahren KMeans, Hierarchisches Clustering, Spectral Clustering und MeanShift.

Die Ergebnisse des Hierarchische Clustering fallen nicht so gut aus, da hier die Cluster nicht horizontal gebildet werden, denn hier wird nicht von einem Zentrum aus gegangen. Stattdessen werden nur die am nächsten beieinander liegenden Werte zusammen geclustert, was teilweise zu vertikalen Clustern führt.

Auch das Spectral Clustering liefert keine guten Ergebnisse, da dieses Cluster nicht anhand der Kompaktness der Werte, sondern anhand ihrer Verbundenheit bildet. Da es sich bei den Werten aber um keinen Graphen handelt, wird die Verbundenheit durch eine Affinitäts-Matrix errechnet.

KMeans liefert gute Ergebnisse, da hier von einem Zentrum aus radial geclustert wird. Somit bilden sich, vor allem bei gleichmäßig verteilten Werten, Cluster welche gut in die vordefinierten drei Bereiche passen.

MeanShift gelingt es mit dem Verschieben des dichtebasierten Kernels größtenteils gute Ergebnisse zu erzielen. Da hier jedoch nicht mit festen Zentren gearbeitet wird fallen die Cluster nicht so stark sphärisch aus wie bei KMeans. Daher kommt es immer Mal wieder dazu, dass ein paar Cluster die gedachten Grenzen der drei Bereiche überschreiten.

Im Vergleich fallen die Ergebnisse von KMeans und MeanShift in allen Intervallen ähnlich aus, doch KMeans gelingt die Unterteilung stärker horizontal und damit besser.

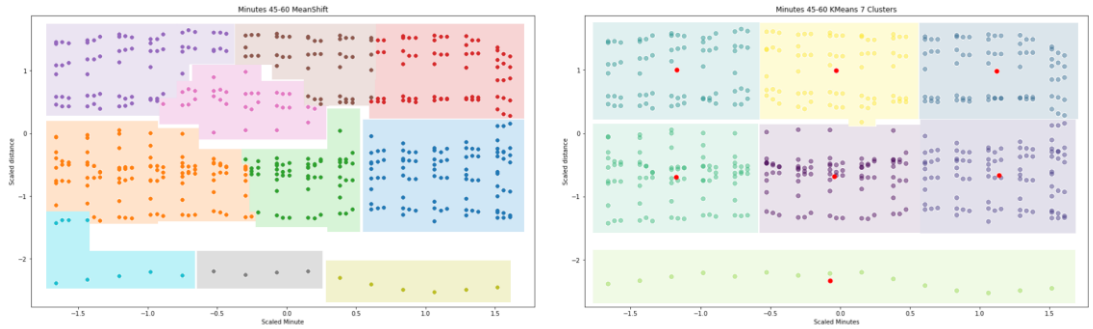

Abbildung 35: Vergleich der von MeanShift und KMeans gebildeten Cluster im Intervall 45-60 (Eigendarstellung)

Zusätzlich kann KMeans diese Ergebnisse mit einer geringeren Anzahl an Clustern erzielen und es läuft schon beim Clustering schneller durch als MeanShift. Es ist also zu erwarten, dass bei dem Klassifikator die Ergebnisse von KMeans schneller erzielt werden als bei MeanShift. In Anbetracht dieser Punkte wird der Klassifikator für die *dfldata* Daten auf Basis des KMeans Clusterings mit sieben Clustern aufgebaut werden.

3 Exploration der DFL-URL Rohdaten

3.1 Aufbau der DFL-Daten

Für die zweite Datenquelle, die DFL-Daten, gibt es zwei unterschiedliche URLs, welche abgefragt werden können, eine ist mit Ereignisdaten gefüllt und die andere mit Positionsdaten. Unter beiden URLs sind XML-Dateien vorzufinden, welche die Daten aller Spieler des angegebenen Spiels speichern. Bei beiden enthält der oberste Tag *PutDataRequest* einen *RequestId*, eine *MessageTime*, und die booleschen Variablen *TransmissionComplete* und *TransmissionSuspended*. Ebenfalls bei beiden vorhanden ist der darunterliegende Tag *MatchStatistic* welcher

Informationen über das abgefragte Spiel enthält, wie den Titel des Spiels *GameTitle*, den Spieltag *MatchDay*, die Match- und Saison-IDs *MatchId* und *SeasonId*, den Status des Spiels *MatchStatus*, die Spielminute *MinuteOfPlay* und noch weitere Informationen. Die beiden URLs unterscheiden sich in diesem Tag nur um eine Variable, die Ereignisdaten enthalten hier die Variable *Result* die das Ergebnis des Spiels enthält und die Positionsdaten enthalten die Variable *Scope*. Darunter sind bei beiden URLs die Tags *TeamStatistic* der zwei teilnehmenden Mannschaften. Beide URLs enthalten dabei den Teamnamen, die Rolle des Teams, die *TeamID* und den *ThreeLetterCode*. In den Ereignisdaten sind hier Werte wie *SubstitutionsIn, CardsYellow, ShotsAtGoalSuccessfull, Assists* usw. enthalten. In den Positionsdaten sind hier Werte wie *Sprints, SprintsBallOur, DistanceCoveredSprints, PlayinTimeSpeedRange1, IntensiveRunsLeftwardGross* usw. enthalten. Unter diesen Tags befinden sich bei beiden URLs für jeden Spieler des Teams der an diesem Tag im Kader stand ein Tag *PlayerStatistic* welcher Informationen über den jeweiligen Spieler liefert. Diese Tags enthalten bei den Positionsdaten unter anderem die Attribute *Sprints, DistanceCovered* und *IntensiveRuns* und bei den Ereignisdaten Attribute wie *shotsAtGoalSum, PassesSuccessfulSum, TacklingGamesWon, Chances, Assists* usw.

Die Attribute, die bei dieser Datenquelle aufgezeichnet werden, sind: *sprints, sprintsNet, dictsnceCovered, distanceCoveredNet, intenseRuns, intenseRunsNet, averageSpeed, maxSpeed, shotsAtGoalSum, passesSuccessfulSum, defClearances, tackAirWon, tackAirLost, tackGroundWon* und *tackGroungLost*. Leider war es nicht möglich, die genauen Beschreibungen dieser herauszufinden.

3.2 Datenbeschaffung/Implementierung

Das geschriebene Python Programm fragt minütlich die Positionsdaten-URL und die Ereignisdaten-URL ab und speichert die Werte für jeden Spieler in jeweils einer CSV-Datei. Um auszuwählen welches Spiel abgefragt wird, muss eine Variable im Code angepasst werden, diese reicht aus, um beide URLs anzupassen. Im Gegensatz zu der Abfrage der Datenbank *dfldata* muss kein weiteres Programm neben diesem für die Abfrage gestartet werden. Das Programm sollte ein paar Minuten vor Spielbeginn gestartet werden, die Aufzeichnung beginnt dann ab Spielbeginn. Die Halbzeitpause wird vom Programm selbständig erfasst und während dieser

werden keine Abfragen getätigt. Bei Beginn der zweiten Halbzeit startet die Aufzeichnung wieder selbstständig und nach Abpfiff des Spiels beendet sich das Programm. Ein weiterer Vorteil bezüglich der minütlichen Abfrage der *dfldata* ist, dass hier die aktuelle Spielminute direkt aus den URLs abgefragt werden kann, somit muss kein implementierter Minuten Zähler mitlaufen. Dadurch werden genauere Abfragen erzielt. Außerdem werden hierbei auch die Nachspielzeiten berücksichtigt.

2Url-Abfrage.py

get_url(var): Bekommt einen String *var* übergeben. Gibt abhängig von *var* entweder die URL der Positionsdaten oder die URL der Ereignisdaten zurück.

get_minute(root): Bekommt eine Root *root* übergeben und gibt den Wert unter *MinuteOfPlay* zurück.

get_match_status(root): Bekommt eine Root *root* übergeben und gibt den Wert unter *MatchStatus* zurück.

get_matchday(root): Bekommt eine Root *root* übergeben und gibt den Wert unter *MatchDay* zurück.

get_team_names(root): Bekommt eine Root *root* übergeben und gibt die Werte unter *TeamNames* beider am Spiel teilnehmender Teams als Tupel zurück.

get_team_codes(root): Bekommt eine Root *root* übergeben und gibt die Werte unter *ThreeLetterCode* beider am Spiel teilnehmender Teams als Tupel zurück.

filter_team(root, teamName): Bekommt eine Root *root* und einen String *teamName* übergeben. Sucht in den Tags unter *root* den Tag welcher die Variable *teamName* enthält und gibt diesen Teil des das Element-Trees zurück.

get_player_dicts_pos(team_tree): Bekommt ein Team entsprechenden Element-Tree *team_tree* übergeben und sammelt für jeden darunterliegenden Spieler-Tag die Werte der gewünschten Attribute in jeweils ein Dictionary *player_dict*. Sammelt alle diese Dictionarys dann in einer Lister *children*. Rückgabewert ist die Liste *children* der Form: [*{Spielername_n: [<sprints des Spielers>, <sprintsNet des Spielers>, <intenseRuns des Spielers>,*

<intenseRunsNet des Spielers>, <distanceCovered des Spielers>, < distanceCoveredNet des Spielers>, <averageSpeed des Spielers>, <maximumSpeed des Spielers>]}, {Spielername_n2: ...}, ...]

get_player_dicts_er(team_tree): Arbeitet wie *get_player_dicts_pos()* gibt jedoch eine Liste der Form: [*{Spielername_n: [<shotsAtGoalSum des Spielers>, <assists des Spielers>, <assistsShotsAtGoal des Spielers>, <passesSuccessfulSum des Spielers>, <defensiveClearances des Spielers>, < tacklesAirWon des Spielers>, <tacklesAirLost des Spielers>, <tacklesGroundWon des Spielers>, <tacklesGroundLost des Spielers>,]}, {Spielername_n2: ...}]* zurück.

initiale_csv(team_tree, matchday, team_code): Bekommt einen Team entsprechenden Element-Tree *team_tree* übergeben, den Spieltag in Form einer Zahl *matchday* und einen String *team_code*. Erstellt für jeden Spieler in *team_tree* eine CSV-Datei des Formats: *Minute, sprints, sprintsNet, intenseRuns, intenseRunsNet, distanceCovered, distanceCoveredNet, averageSpeed, maxSpeed, shotsAtGoalSum, assists, assistsShotsAtGoal, passesSuccessfulSum, defClearances, tackAirWon, tackAirLost, tackGroundWon, tackGroundLost*. Der Name der CSV-Datei wird mithilfe des Spielernamens, dem Spieltag und des Teamcodes nach dem Muster: *<team_code>_<Spielername>_ST<matchday>.csv* erstellt.

csv_fuellen(team_dict_pos, team_dict_er, matchday, minute, team_code): Bekommt die Team-Dictionarys der Positionsdaten *team_dict_pos* und der Ereignisdaten *team_dict_er* übergeben, sowie die Zahlen *matchday* und *minute* und den String *team_code*. Erstellt ein leeres Dictionary *attDict*. Sammelt zuerst pro Spieler die Attribute aus *team_dict_pos* in *attDict* unter dem Key des Spielernamens ein und fügt diesen dann an den entsprechenden Spielernamen die Attribute aus *team_dict_er* hinzu. Füllt dann für jeden Spieler die CSV-Datei namens *<team-code>_<Spielername>_ST<matchday>.csv*, mit den Attributen aus *attDict* unter dem Key des Spielernamens.

Main Methode: Die Abfrage läuft über zwei While-Schleifen ab, vor den beiden Schleifen werden die Variablen *init_parse*, *status* und *first* initialisiert. Die Variable *init_parse* bekommt dabei den Wert *„Running“* zugewiesen und die Variable *status* den Wert *„fistHalf“*. In der ersten While- Schleife wird überprüft, ob das Spiel beendet wurde, in der zweiten wird überprüft, ob das Spiel noch in der ersten oder zweiten Halbzeit läuft. Innerhalb der zweiten While-

Schleife werden, jedes Mal, wenn sich die Minuten-Anzeige in den URLs ändert, die CSV-Dateien der Spieler befüllt. Mit der Variable *first* wird überprüft, ob es sich um den ersten Durchlauf handelt und somit neue CSV-Dateien angelegt werden müssen. Die Überprüfung, ob das Spiel noch läuft, erfolgt mit der Variablen *status* welche am Anfang initialisiert wurde, diese wird innerhalb der zweiten While-Schleife aus den URLs abgefragt. Wenn es zur Halbzeitpause kommt, enthält die Abfrage vom Spielstatus den Wert *„half“* somit wird auch *status* auf *„half“* gesetzt und die zweite While-Schleife wird nicht mehr durchlaufen. Stattdessen wird nun die erste While-Schleife durchlaufen, welche *status* ebenfalls aus den URLs abfragt. Bei Beginn der zweiten Halbzeit kann somit die zweite Schleife weiter durchlaufen werden. Zur Überprüfung, ob das Spiel beendet wurde, läuft in der zweiten Schleife eine Abfrage, welche die Variable *init_parse* gegebenenfalls auf *„finalWhistle“* setzt. Wenn das Spiel beendet ist, wird *status* auf *„finalWhistle“* gesetzt und die zweite Schleife wird beendet, die erste Schleife wird daraufhin auch nicht länger durchlaufen, da ihre Durchlaufbedingung voraussetzt, dass *init_parse* ungleich *„finalWhistle“* ist.

3.3 Clustering

Für das Clustering werden Daten von den Spieltagen 28, 29, 30 und 32 betrachtet, dabei werden nur Spieler in Betracht gezogen, die in der Startelf einer Mannschaft in der Verteidigung standen. Es werden hierbei Verteidiger verschiedener Mannschaften aufgezeichnet, da ansonsten zu wenig Daten vorhanden sein würden. Aufgrund der vorherigen Ergebnisse aus den *dfldata* Daten werden Spieler, die für Verteidiger eingewechselt wurden, nicht mit einbezogen. Der Datensatz besteht insgesamt aus 3449 Zeilen und wird erneut in sechs 15 Minuten Intervalle unterteilt und mit dem *StandardScaler* skaliert. Es wird primär am Beispiel des Attributes *sprints* geclustert, aber auch an den Attributen *distanceCoveredNet* und *intenseRunsNet*.

3.3.1 Vorgegebene Anzahl an Clustern

KMeans

Die Elbow-Methode ergab für KMeans eine ideale Clusteranzahl von drei. Doch mit drei Clustern ließ sich keine logische Unterteilung erzielen. Daher wurde die Anzahl an Clustern erhöht. Bei sechs Clustern ließen sich die drei Bereiche zwar schon in allen Intervallen unterscheiden,

doch die Ergebnisse waren noch besser mit mehr Clustern.

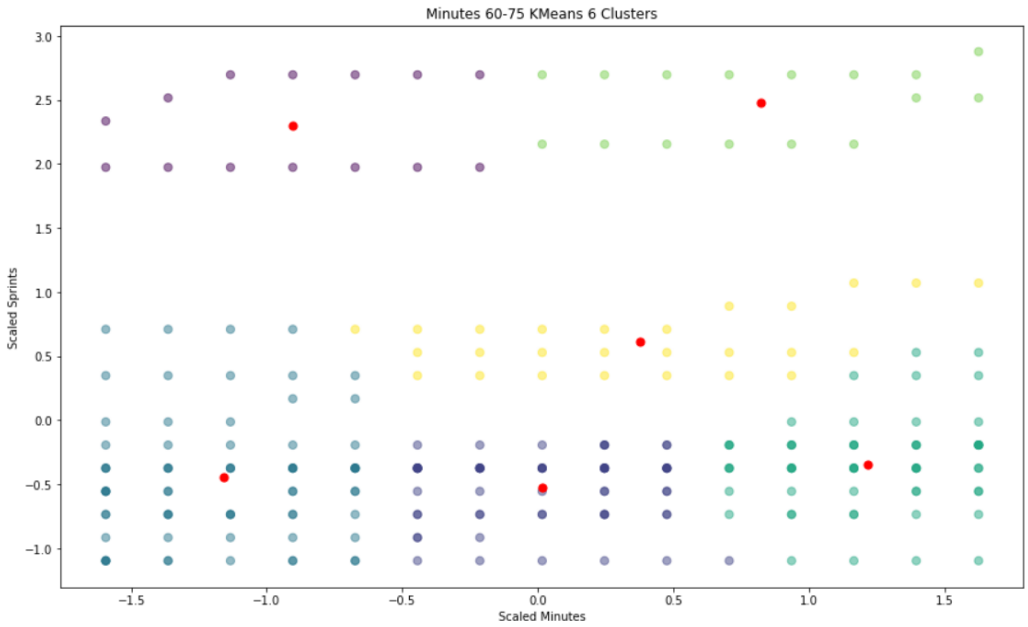

Abbildung 36: KMeans Clustering des Intervalls 60-75 von *sprints* mit sechs Clustern (Eigendarstellung)

Beispielsweise wären beim Intervall 60-75 die beiden oberen Cluster dem Bereich „Gut“ zuzuordnen, die unteren drei dem Bereich „Schlecht“ und das gelbe Cluster in der Mitte dem Bereich „Durchschnittlich“. Doch der mittlere Bereich trennt die „Guten“ und „Schlechten“ Werte nicht vollständig voneinander. Daher könnte derselbe Wert einmal als „Durchschnittlich“ und ein anderes Mal als „Schlecht“ angesehen werden, was aufgrund der geringen Anzahl an vergangenen Minuten nicht logisch wäre.

Die Ergebnisse mit sieben oder auch acht Clustern hingegen weisen dieses Problem nicht auf. Diese Ergebnisse wiesen nur minimale Unterschiede untereinander auf, mit acht Clustern fielen diese insgesamt noch ein wenig besser aus. Auch Attributs-übergreifend waren die Ergebnisse

mit acht Clustern etwas besser, daher wurde entschieden mit acht Clustern zu arbeiten.

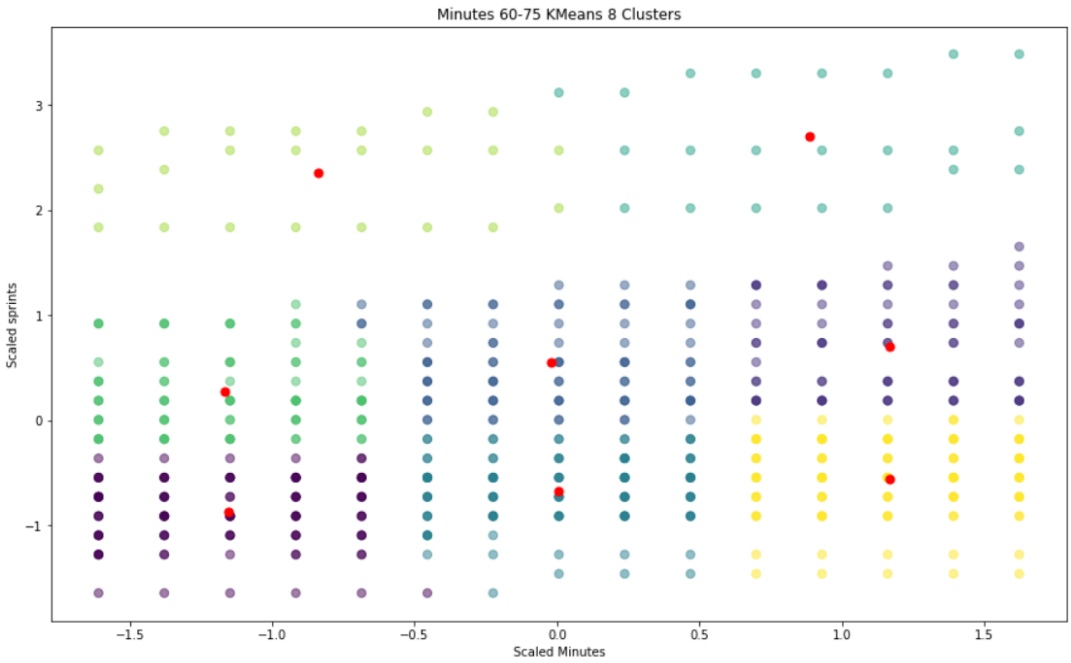

Abbildung 37: KMeans Clustering des Intervalls 60-75 von *sprints* mit acht Clustern (Eigendarstellung)

Mit acht Clustern wird bei jedem Intervall eine größtenteils horizontale Trennung in die Bereiche „Gut“, „Schlecht“ und „Durchschnittlich“ ermöglicht. Dies liegt wohl vor allem an der Tatsache, dass bei KMeans die Cluster sphärisch um einen Mittelpunkt gebildet werden. Auch das Clustern des Attributes *distanceCoveredNet* lieferte in jedem Intervall angemessene, größtenteils sogar gute Ergebnisse. Das einzige Intervall, welches nur ein angemessenes Ergebnis hervorbrachte, war das Intervall 0-15. Bei Betrachtung dieses kommt die Frage auf, ob nicht doch eine Ausreißer Behandlung vorgenommen werden sollte. Allerdings ist dies das einzige beobachtete Intervall, welches eine derartige Verteilung der Werte aufzeigt.

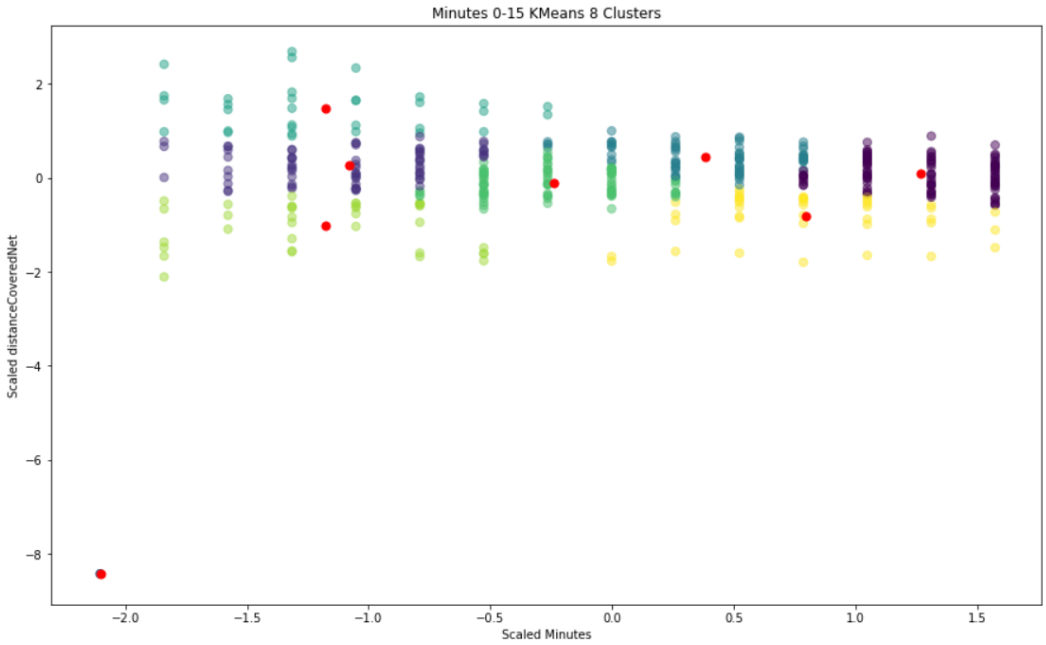

Abbildung 38: KMeans Clustering des Intervalls 0-15 von *distanceCoveredNet* mit acht Clustern (Eigendarstellung)

In diesem ist ein einziger Wert weit von dem Rest der Werte gelegen. Aufgrund der festen Anzahl an Clustern beansprucht dieser daher ein gesamtes Cluster für sich allein und die restlichen Werte werden auf sieben Cluster verteilt. Zwar bleibt das Ausmaß der Verteilung in einem annehmbaren Rahmen, da von den Übrigen sieben Clustern die unteren zwei dem Bereich „Schlecht“ zugeordnet werden können die mittleren vier dem Bereich „Durchschnittlich“ und das oberste Cluster dem Bereich „Gut“, doch es ist anzunehmen, dass das Ergebnis ohne diesen Ausreißer noch besser ausgefallen wäre.

Aufgrund der kontinuierlich guten Ergebnisse des Clusterings wurde auch auf dem Attribut *intenseRunsNet* geclustert. Wie schon bei den vorherigen beiden Attributen war hier der Großteil der Ergebnisse gut, da die Cluster sich derartig formten, dass die Bereiche hauptsächlich horizontal voneinander getrennt wurden.

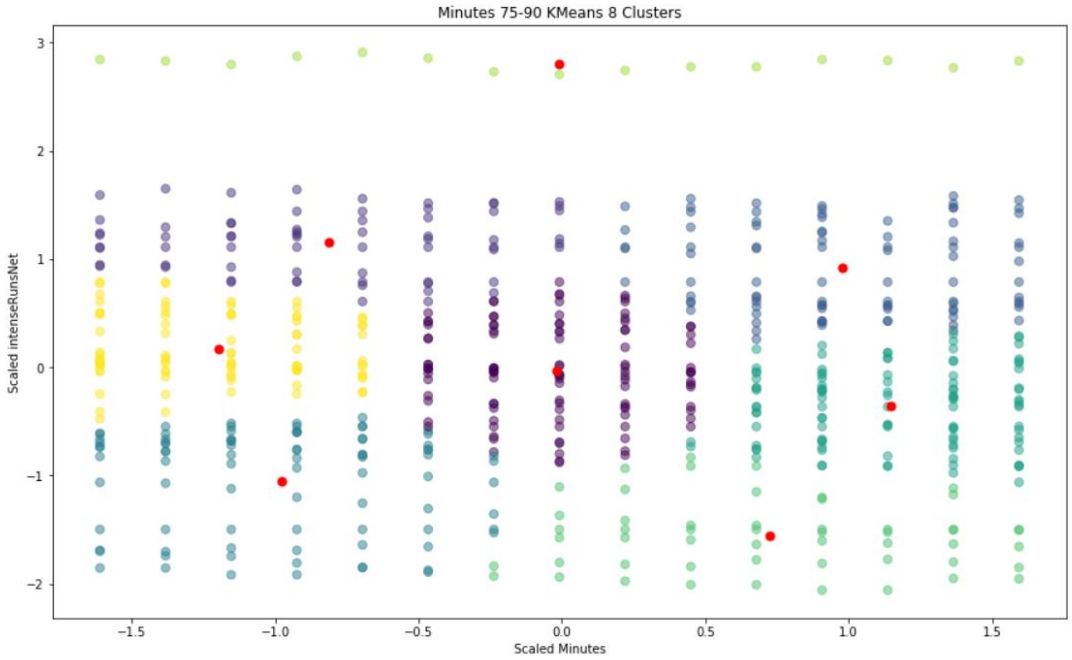

Abbildung 39: KMeans Clustering des Intervalls 75-90 von *intenseRunsNet* mit acht Clustern (Eigendarstellung)

Daher ist festzuhalten, dass die Ergebnisse mit KMeans und einer Clusteranzahl von acht sowohl Intervallübergreifend als auch Attributs-übergreifend gute Ergebnisse liefern und dass es im schlimmsten Fall zu immer noch vertretbaren Ergebnissen kommt.

Hierarchisches Clustering

Da acht Cluster bei KMeans gute Ergebnisse geliefert hatten, wurde auch beim Hierarchischen Clustering mit acht Clustern gearbeitet. Bei dem Attribut *sprints* entstehen in jedem Intervall annehmbare Ergebnisse.

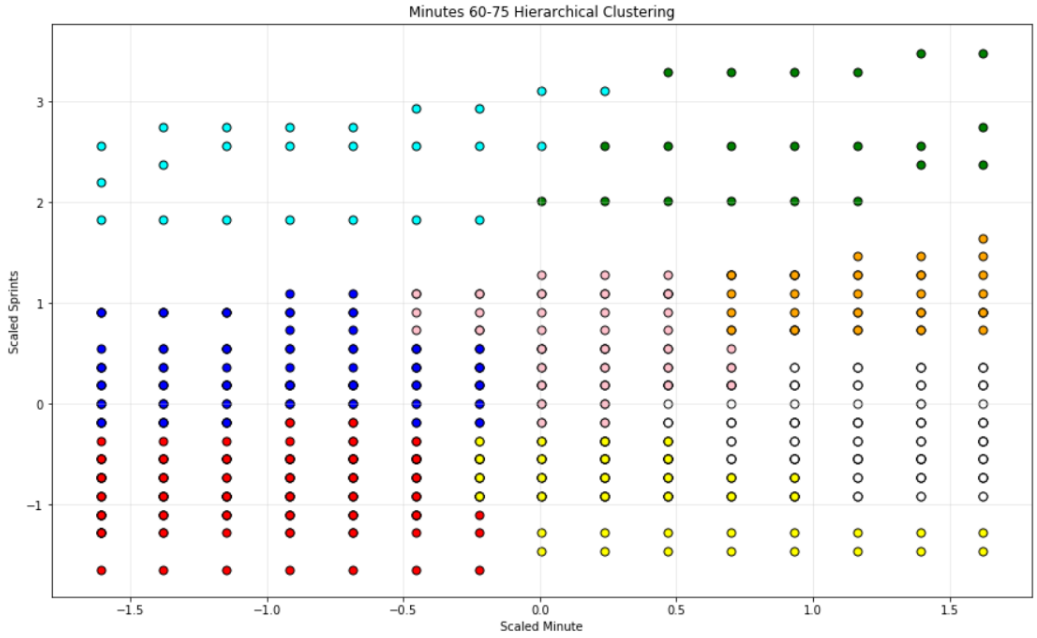

Abbildung 40: Hierarchisches Clustering des Intervalls 60-75 von *sprints* mit acht Clustern (Eigendarstellung)

Wie bei dem Intervall 60-75, hier werden die Bereiche zwar horizontal voneinander getrennt, doch die Bereiche verlaufen stärker ineinander als bei KMeans. Bei dem Attribut *distanceCoveredNet* sind gemischte Ergebnisse vorhanden. Einige Intervalle wie das Intervall 75-90 werden gut geclustert, die horizontale Trennung der Bereiche ist hierbei einfach möglich dementsprechend können die Cluster ihrem Bereich sinnvoll zugeordnet werden. Bei anderen Intervallen werden allerdings weniger sinnvolle Cluster gebildet, wie bei dem Intervall 30-45.

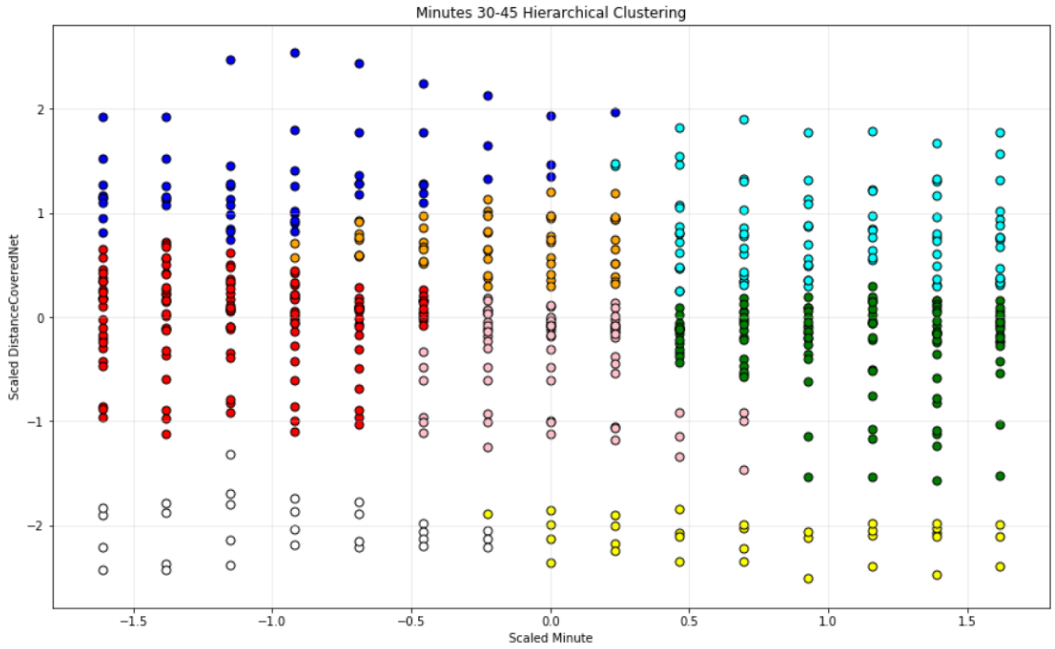

Abbildung 41: Hierarchisches Clustering des Intervalls 30-45 von *distanceCoveredNet* mit acht Clustern (Eigendarstellung)

Hier bildet das hellblaue Cluster ein Problem, da dieses Cluster logisch dem Bereich „Gut“ zugeordnet werden müsste, doch aufgrund der Tatsache, dass sich viele seiner Werte tiefer in den X-Werten befinden, könnte dieses Cluster später vom Klassifikator aber als „Durchschnittlich“ angesehen werden. Dementsprechend könnten Werte die als „Gut“ klassifiziert werden sollten fälschlicher Weise als „Durchschnittlich“ klassifiziert werden. Dieses Problem ist beim Clustern des Attributes *intenseRunsNet* ebenfalls und sogar vermehrt zu beobachten. Hier könnte dieses Problem in den Intervallen 0-15, 30-45, 60-75 und 75-90 in dieser oder einer anderen Form auftreten.

Daher bildet das Hierarchische Clustering in dem meisten Fällen zwar akzeptable Cluster, in einigen Fällen sogar gute, doch es werden deutlich mehr schlechte Cluster gebildet als bei KMeans.

BIRCH

Auch hier wurde aufbauend auf den vorherigen guten Ergebnissen mit acht Clustern weiter mit acht Clustern gearbeitet. Dabei wurde der Parameter *threshold* auf 0.35 gesetzt und der

Parameter *branching-factor* auf 5. Es kam zu gemischten Ergebnissen, die Intervalle 0-15, 30-45 und 60-75 wurden akzeptabel geclustert, sodass eine sinnvolle Unterteilung in die jeweiligen Bereiche größtenteils möglich wäre,

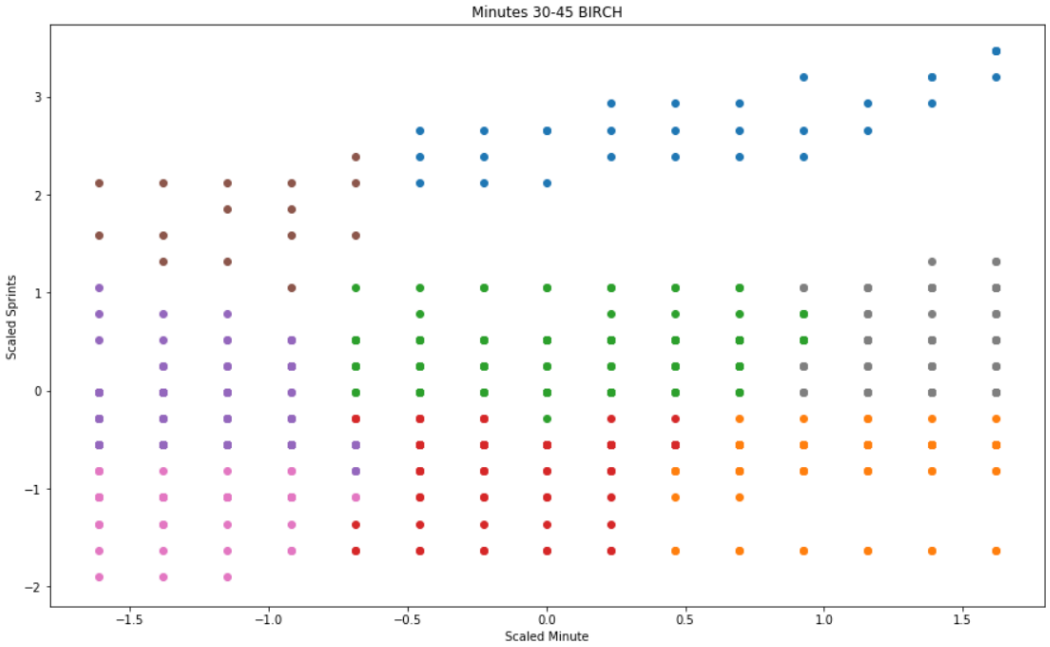

Abbildung 42: BIRCH Clustering des Intervalls 30-45 von *sprints* mit acht Clustern, *branching-factor = 5, threshold = 0.35* (Eigendarstellung)

während bei den Intervallen 15-30, 45-60 und 75-90 die Cluster weniger sinnvoll unterteilt werden konnten. Hier kommt, wie auch beim Hierarchischen Clustering, das Problem auf, dass einige Werte dem falschen Bereich zugeordnet werden würden, da die Cluster zu stark ineinander verschwimmen.

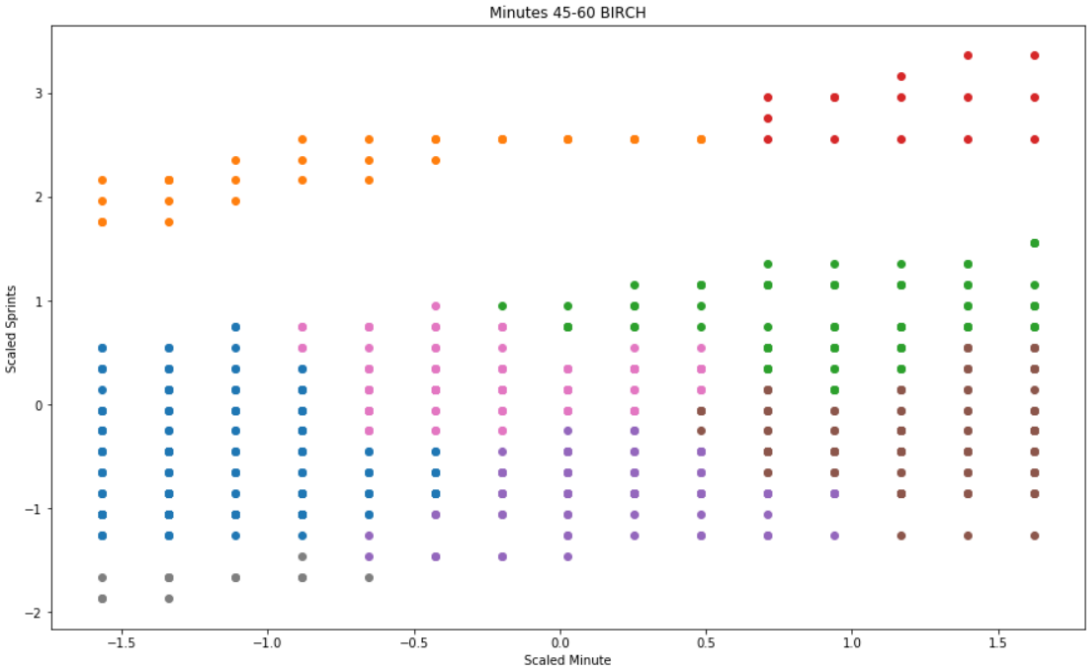

Abbildung 43: BIRCH Clustering des Intervalls 45-60 von *sprints* mit acht Clustern, *branching-factor = 5, threshold = 0.35* (Eigendarstellung)

Hier überbrückt beispielsweise das blaue Cluster die Bereiche „Schlecht“ und „Durchschnittlich. Bei einer Anpassung des Parameters *branching-factor* auf 7 wird das Intervall 0-15 gut geclustert, die Intervalle 15-30, 45-60 und 60-75 werden akzeptabel geclustert und die Intervalle 30-45 und 75-90 werden nicht sinnvoll geclustert.

Eine weitere Anpassung des Parameters *threshold* auf 0.5 führte dazu, dass die Intervalle 15-30, 45-60 und 60-75 das bisher, für das BIRCH Verfahren, beste Ergebnis erzielten.

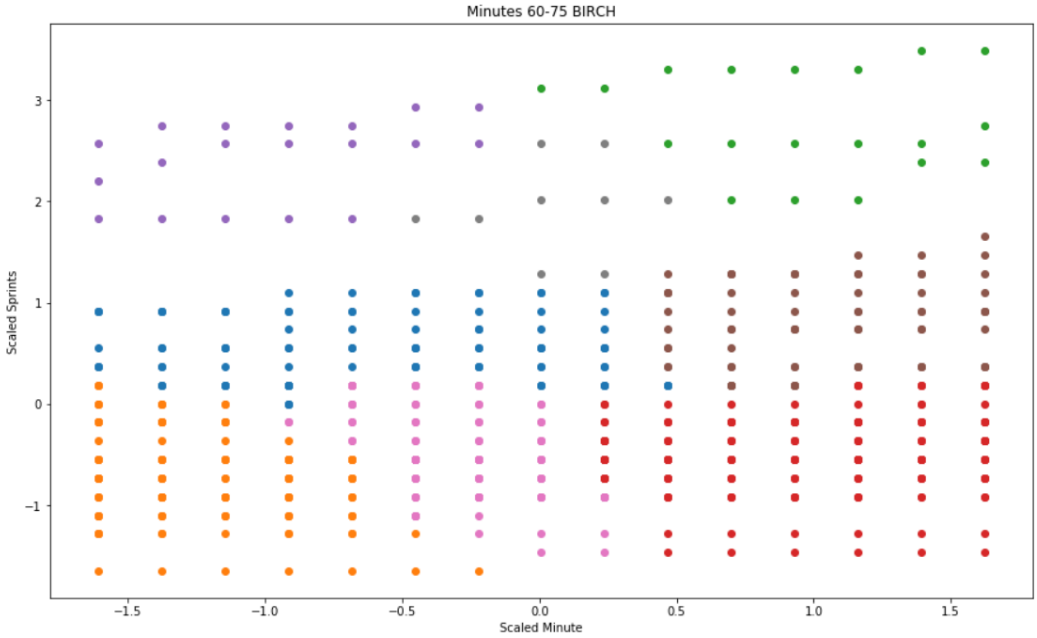

Abbildung 44: BIRCH Clustering des Intervalls 60-75 von *sprints* mit acht Clustern, *branching-factor = 7, threshold = 0.5* (Eigendarstellung)

In diesen waren nur sehr wenige oder gar keine Übergänge der Cluster in mehrere Bereiche zu beobachten. Die Intervalle 30-45 und 75-90 zeigten jedoch deutlich schlechtere Ergebnisse auf, in welchen die Bereiche nur schlecht horizontal voneinander getrennt werden können.

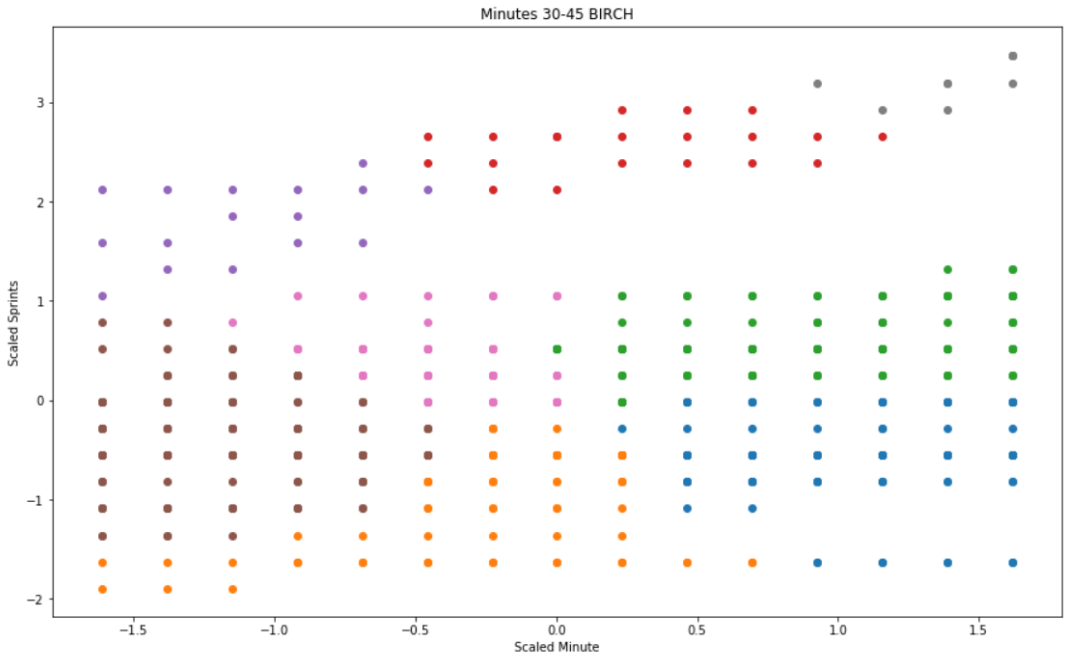

Abbildung 45: BIRCH Clustering des Intervalls 30-45 von *sprints* mit acht Clustern, *branching-factor = 7, threshold = 0.5* (Eigendarstellung)

So würde im Intervall 30-45 das orange und das blaue Cluster dem Bereich „Schlecht“ zugeordnet werden und das rosa und das grüne Cluster dem Bereich „Durchschnittlich“. Das braune Cluster jedoch könnte beiden Bereichen „Durchschnittlich“ oder „Schlecht“ zugeordnet werden, wobei es Werte beinhalten würde, die auf Höhe des Bereichs „Durchschnittlich“ und „Schlecht“ liegen. Dies könnte zu falschen Klassifikationen führen.

Um ein angemessenes Ergebnis für alle Intervalle zu erzielen, bedarf es schon innerhalb des Attributes *sprints* Anpassungen der Parameter *threshold* und *branching-factor* von Intervall zu Intervall. Selbst wenn die Anpassungen dementsprechend gemacht wurden, sind die Ergebnisse vorheriger Clustering-Verfahren besser als die von BIRCH.

Spectral Clustering

Bei dem Spectral Clustering wurde, wie bei den vorherigen Verfahren, mit acht Clustern gearbeitet. Es kommt in allen Intervallen zu angemessenen Ergebnissen, die gebildeten Cluster verschwimmen zwar ab und zu ineinander, jedoch nicht zu stark.

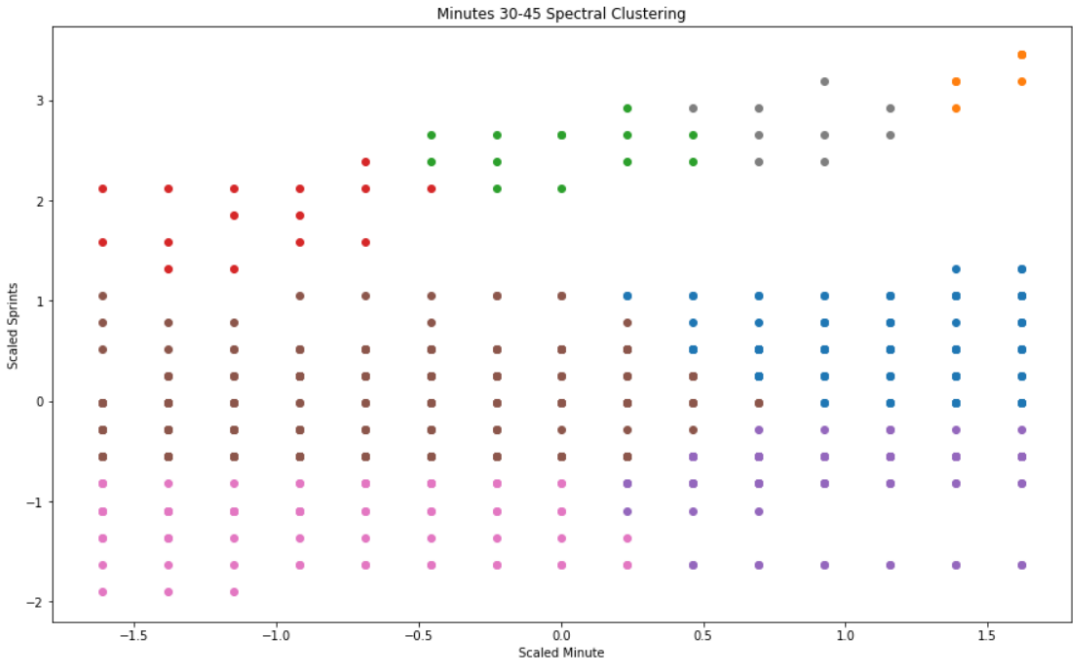

Abbildung 46: Spectral Clustering des Intervalls 30-45 von *sprints* mit acht Clustern (Eigendarstellung)

Bei dem Intervall 30-45 ist zu erkennen, dass das lila Cluster sich einen kleinen Wertebereich mit dem braunen Cluster teilt, wobei das braune Cluster dem Bereich „Durchschnittlich“ und das lila Cluster dem Bereich „Schlecht“ zugeordnet werden würde. Dieser geteilte Wertebereich ist jedoch eher klein und daher würde sich eine Fehlklassifikation auf die Werte des lila Clusters beschränken, welche zwar durchschnittlich sein würden, jedoch als „Schlecht“ klassifiziert werden würden.

In dem Intervall 75-90 kommt es sogar zu einem sehr guten Ergebnis, in diesem werden die Cluster stark horizontal voneinander getrennt.

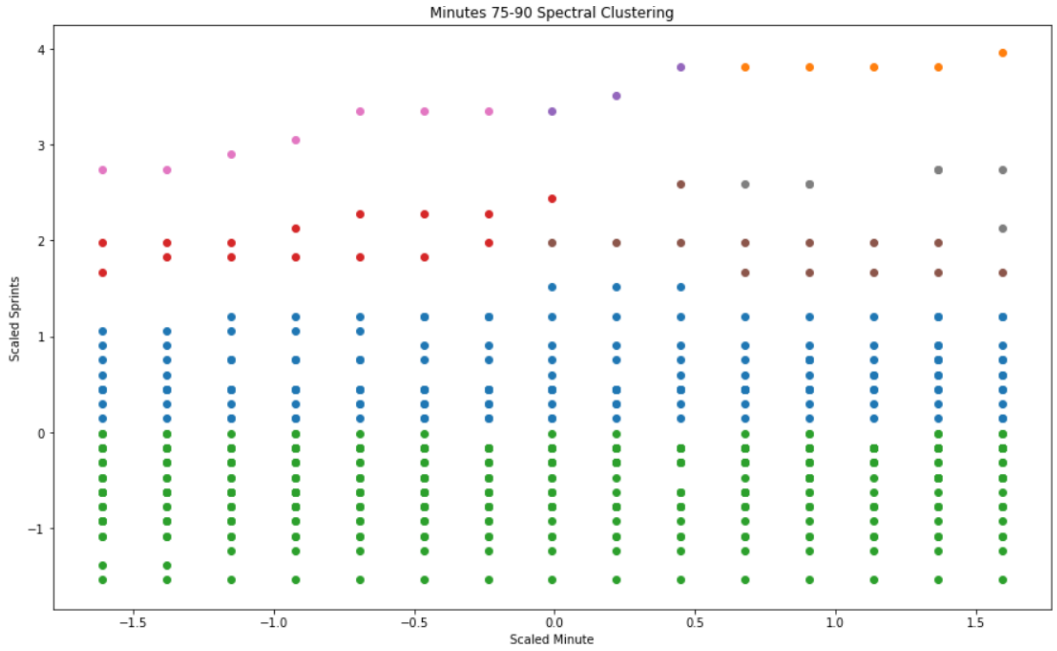

Abbildung 47: Spectral Clustering des Intervalls 75-90 von *sprints* mit acht Clustern (Eigendarstellung)

Dadurch würde es hier zu sehr wenig Fehlklassifikationen kommen. Eine logische Unterteilung würde das grüne Cluster dem Bereich „Schlecht“ zuordnen und das blaue Cluster dem Bereich „Durchschnittlich“. Das braune, rote und graue Cluster würden alle gemeinsam entweder dem Bereich „Durchschnittlich“ oder „Gut“ zugeordnet werden und die restlichen Cluster würden dem Bereich „Gut“ zugeordnet werden.

Bei dem Attribut *distanceCoveredNet* fallen die Ergebnisse etwas schlechter aus. Hier brechen immer wieder einige Cluster durch die gedachten horizontalen Grenzen der Bereiche.

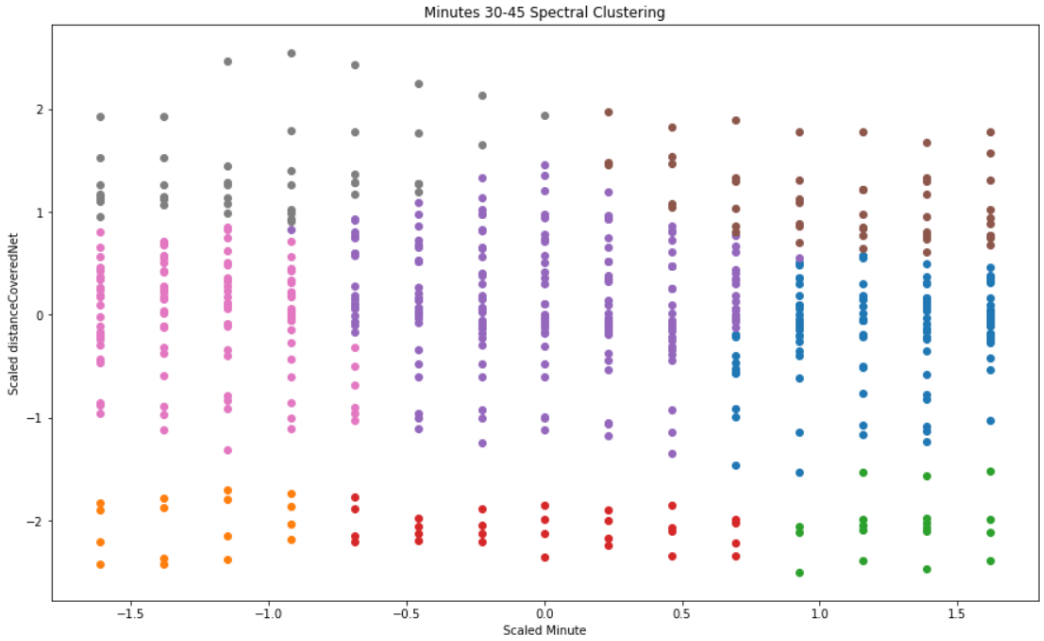

Abbildung 48: Spectral Clustering des Intervalls 30-45 von *distanceCoveredNet* mit acht Clustern (Eigendarstellung)

So wie in dem Intervall 30-45, hier bircht das lilane Cluster, welches logisch dem Bereich „Durchschnittlich“ zugeordnet werden würde, mit seinen Werten in den Bereich „Gut“. Vor allem wird dieses Problem aber bei dem Intervall 45-60 deutlich.

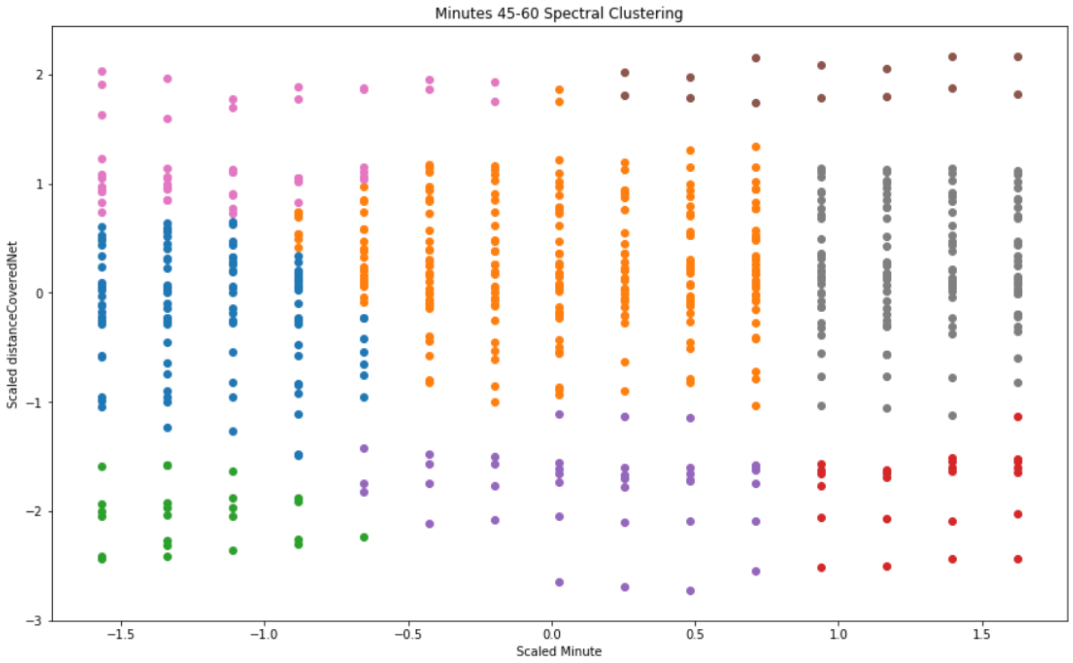

Abbildung 49: Spectral Clustering des Intervalls 45-60 von *distanceCoveredNet* mit acht Clustern (Eigendarstellung)

Hier reicht das pinke Cluster, welches logisch dem Bereich „Gut“ zugeordnet weden würde, mit seinen Werten in den Bereich „Duchschnittlich“ herrein. Dadurch könnte es später zu falschen Klassifizierungen kommen, in welchen ein Wert fälschlicher Weise als „Gut“ klassifiziert wird obwohl er „Durchschnittlich“ sein sollte. Beim Spectral Clustering werden daher größtenteils akzeptable Ergebnisse erzeilt, jedoch kommt es zwischendurch auch zu schlechten Ergebnissen.

3.3.2 Selbst bestimmende Anzahl an Clustern

DBSCAN

Mit DBSCAN ist es praktisch unmöglich mit demselben Wert der Parameter *eps* und *min_samples* gute oder auch nur angemessene Ergebnisse in allen Intervallen zu erzielen, denn auch kleine Unterschiede des Parameters *eps* führen zu großen Veränderungen der Clusterung. So wird das Intervall 0-15 mit dem Parametern *eps* = 0.75 und *min_samples* = 5 gut geclustert, indem jedes Cluster eine horizontale Reihe bildet. Dadurch ist es einfach die Bereiche horizontal zu trennen und Fehlklassifizierungen sind ausgeschlossen. Außerdem wird hier

nur ein Wert als Ausreißer angesehen.

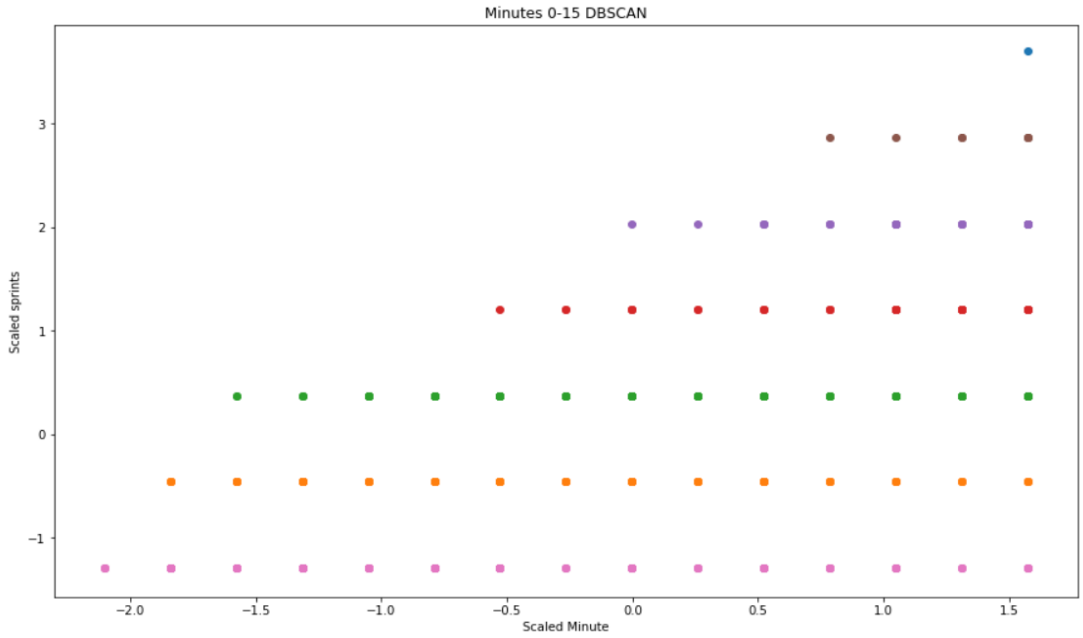

Abbildung 50: DBSCAN Clustering des Intervalls 0-15 von *sprints*, *eps = 0.75*, *min_samples = 5* (Eigendarstellung)

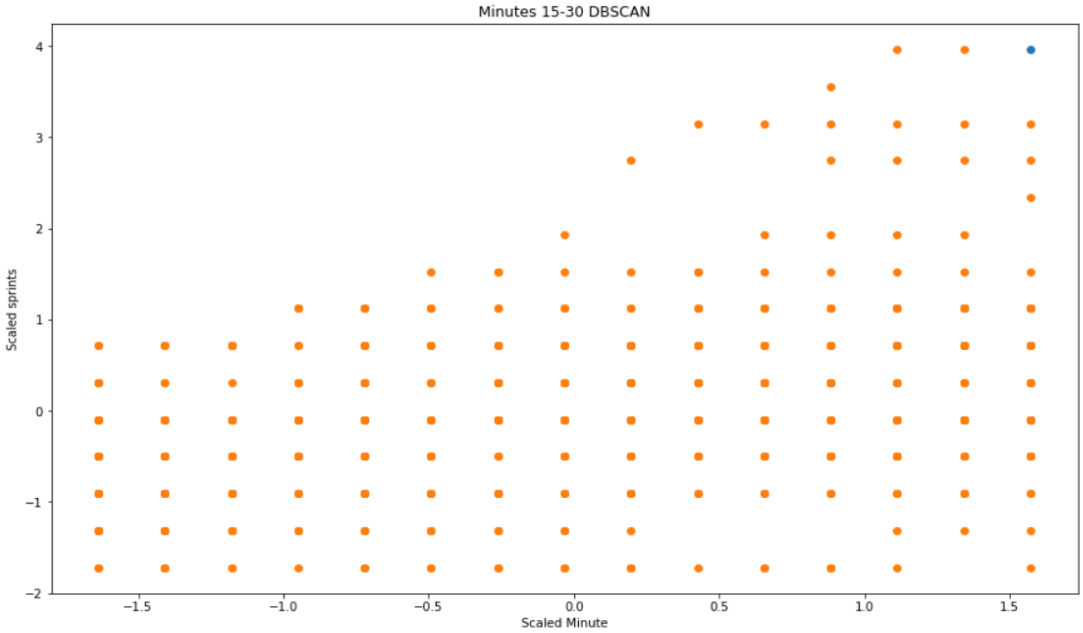

Abbildung 51: DBSCAN Clustering des Intervalls 15-30 von *sprints*, *eps = 0.75*, *min_samples = 5* (Eigendarstellung)

Für das Intervall 15-30 sind dieselben Parameter jedoch schon nicht mehr passend. Hier fällt das Ergebnis sehr schlecht aus, denn offensichtlich ist dieser *eps* Wert zu hoch für dieses Intervall, sodass alle Werte mit Ausnahme eines Ausreißers als ein Cluster angesehen werden.

Da DBSCAN ein dichtebasiertes Verfahren ist und die Werte der verschiedenen Intervalle und der verschiedenen Attribute in unterschiedlichen Abständen voneinander entfernt liegen, ist es nicht möglich mit gleichbleibenden Parametern gute Ergebnisse zu erzielen. Da die Parameter jedoch nicht von Intervall zu Intervall angepasst werden sollen, wird dieses Verfahren daher ausgeschlossen werden müssen.

Affinity Propagation

Bei der Affinity Propagation wurde vorerst mit den Parametern *damping* = 0.75 und *preference* = -10 gearbeitet. Dabei kommt es zu gemischten Ergebnissen. Die Intervalle 15-30 und 60-75 bilden sinnvolle Cluster, in welchen die Bereiche „Gut“, „Schlecht“ und „Durchschnittlich“ horizontal, ohne stark ineinander zu verschwimmen, getrennt werden können. Das Intervall 0-15 wird jedoch nur angemessen geclustert, hier kommt

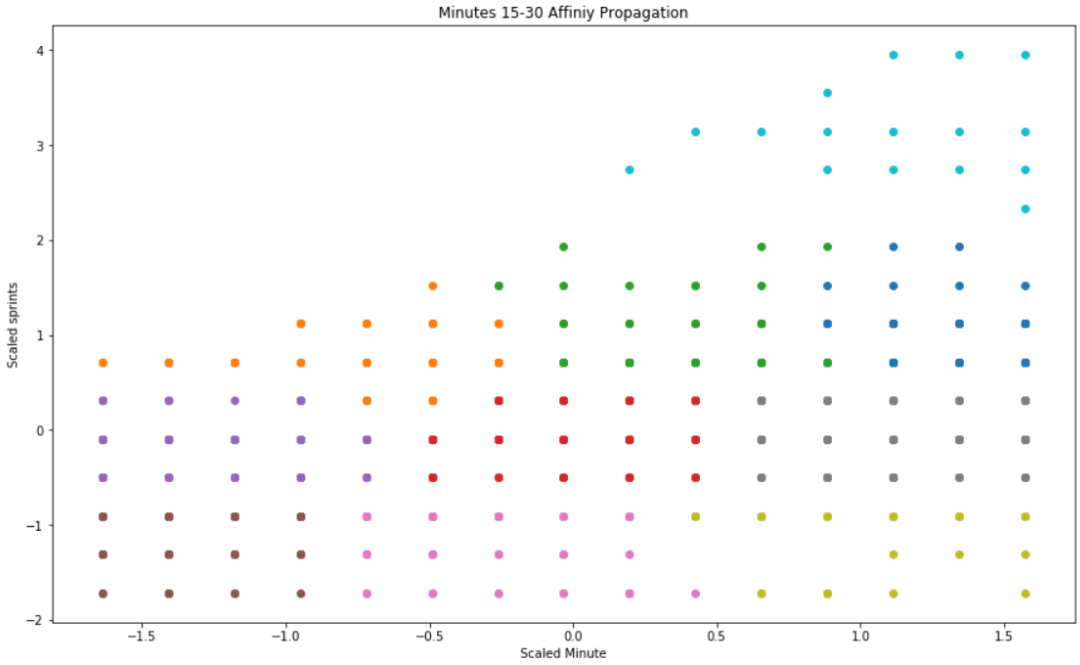

Abbildung 52: Affinity Propagation Clustering des Intervalls 15-30 von *sprints*, *damping = 0.75, preference = -10* (Eigendarstellung)

es dazu, dass die Bereiche leicht ineinander übergehen, was jedoch in einem akzeptablen Rahmen bleibt. Die Intervalle 30-45, 45-60 und 75-90 werden jedoch schlechter geclustert. Hier kommt es dazu, dass die Bereiche stark ineinander übergehen oder die Trennung der Bereiche schräg verläuft.

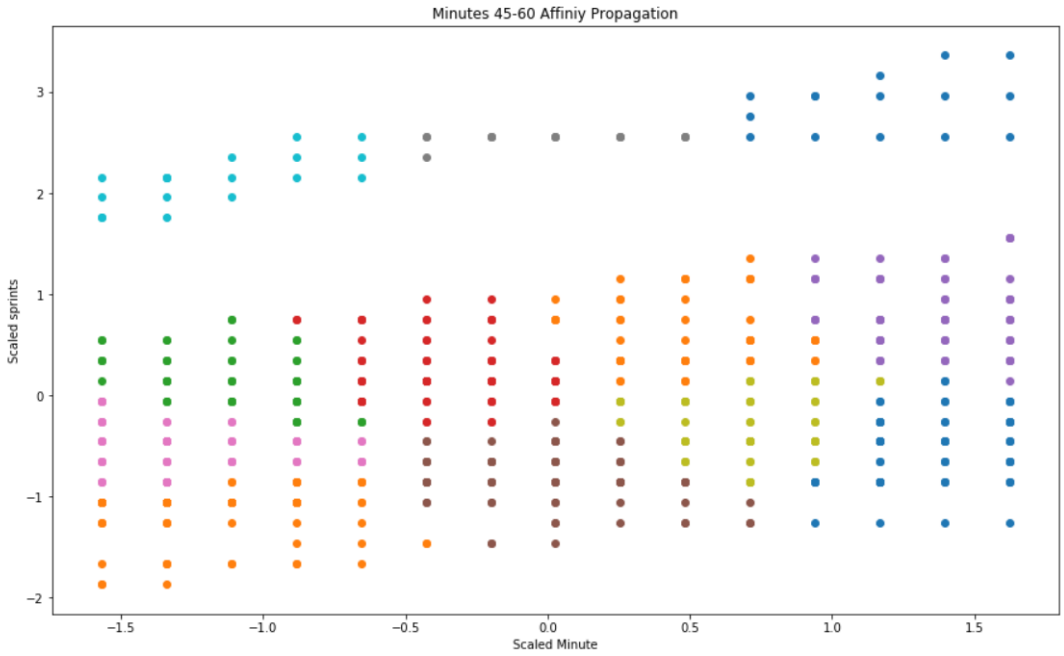

Abbildung 53: Affinity Propagation Clustering des Intervalls 45-60 von *sprints*, *damping = 0.75*, *preference = -10* (Eigendarstellung)

Hierbei bildet die schräge Trennung ein Problem, da es durch diese dazu kommt, dass der Abschnitt, den sich die Bereiche „Durchschnittlich“ und „Schlecht“ teilen, relativ groß ausfällt, wodurch es öfter zu Fehlklassifikationen kommen kann.

Eine Anpassung des Parameters *preference* auf -20 führt dazu, dass sich diese schlechten Ergebnisse verbessern, ohne dass die übrigen Ergebnisse dadurch zu stark negativ beeinflusst werden. Nur das Intervall 0-15 wird etwas in Mitleidenschaft gezogen, dies bleibt jedoch akzeptabel. Diese Parameter führen auch bei dem Attribut *distanceCoveredNet* größtenteils zu akzeptablen Ergebnissen, jedoch kommt es zwischendurch auch zu schlechten Ergebnissen, wie im Intervall 60-75. Hier verschwimmt das orange Cluster des Bereichs „Durchschnittlich“ leicht in den Bereich „Gut“ und das graue sowie das grüne Cluster des Bereiches „Gut“ verschwimmen stark in den Bereich „Durchschnittlich“.

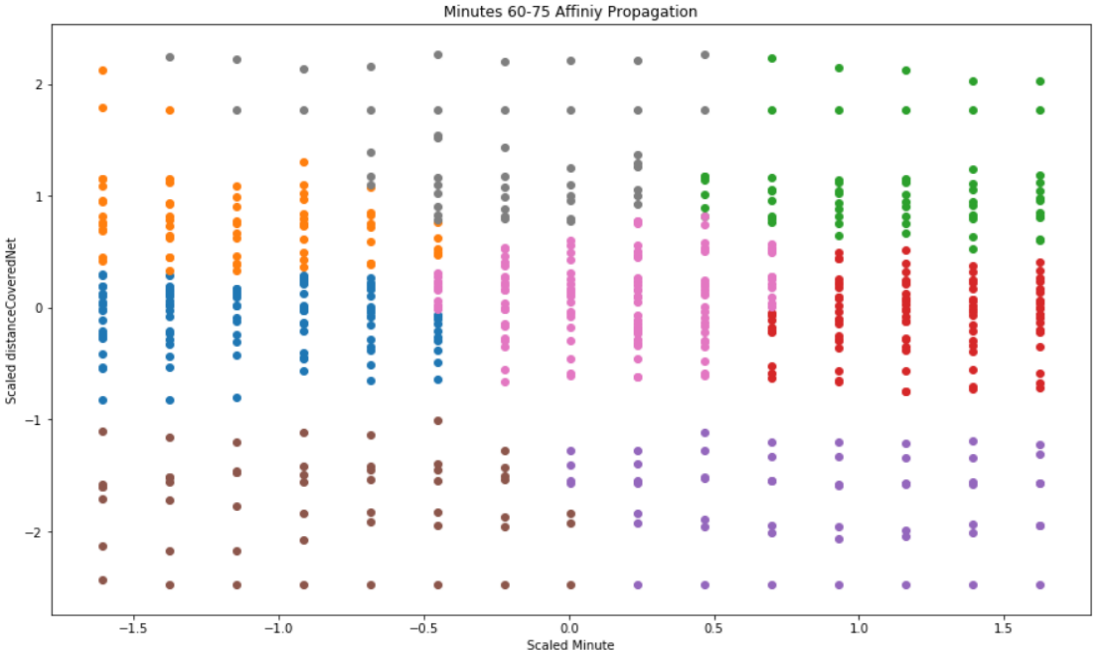

Abbildung 54: Affinity Propagation Clustering des Intervalls 60-75 von *distanceCovered-Net, damping = 0.75, preference = -20* (Eigendarstellung)

Ähnliches ist für das Attribut *intenseRunsNet* in den Intervallen 30-45 und 60-75 beobachten.

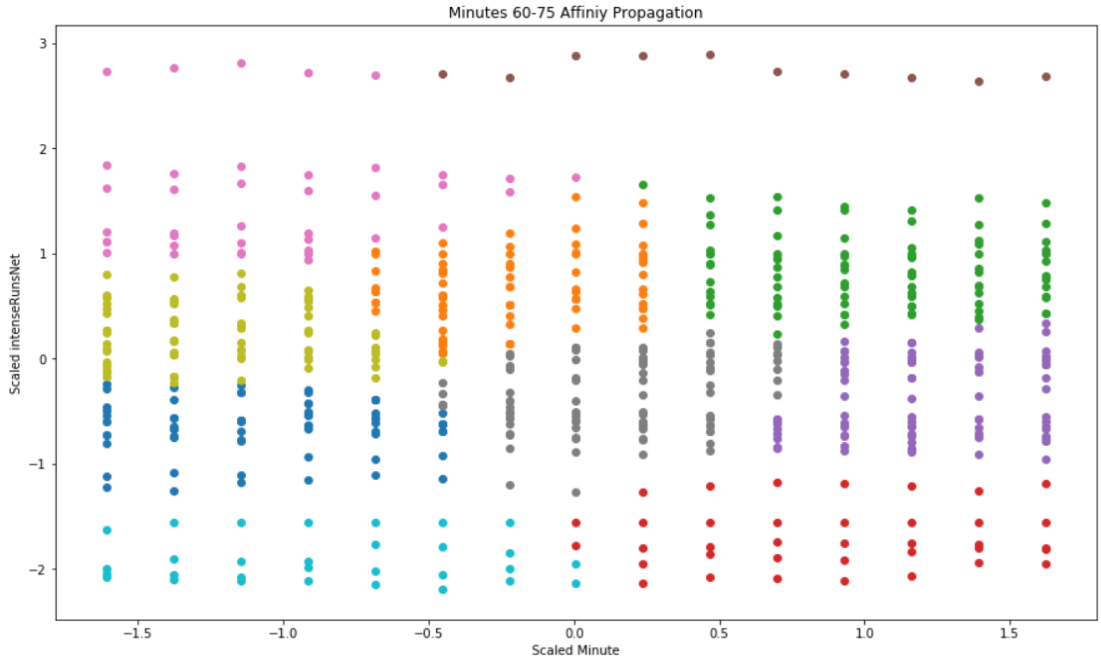

Abbildung 55: Affinity Propagation Clustering des Intervalls 60-75 von *intenseRunsNet, damping = 0.75, preference = -20* (Eigendarstellung)

Allgemein sind bei Affinity Propagation zwar auch ohne Anpassungen der Parameter Intervall-übergreifend und Attributs-übergreifend größtenteils akzeptable Ergebnisse zu erzielen, hierbei fallen einige Ergebnisse jedoch nicht gut aus. Daher müsste falls dieses Verfahren zur Klassifikation genutzt werden sollte, auch mit falschen Einschätzungen gerechnet werden.

MeanShift

Das Clustern mit MeanShift wurde vorerst mit dem Parameter *quantile* von 0.15 durchgeführt. Dies führte in allen Intervallen außer dem Intervall 0-15 zu schlechten Ergebnissen, wobei das Ergebnis des Intervalls 0-15 nur akzeptabel und nicht gut war. Bei einem Großteil der Intervalle wurde mit diesem Parameter nur eine Unterteilung in zwei Bereiche ermöglicht, wie beim Intervall 30-45.

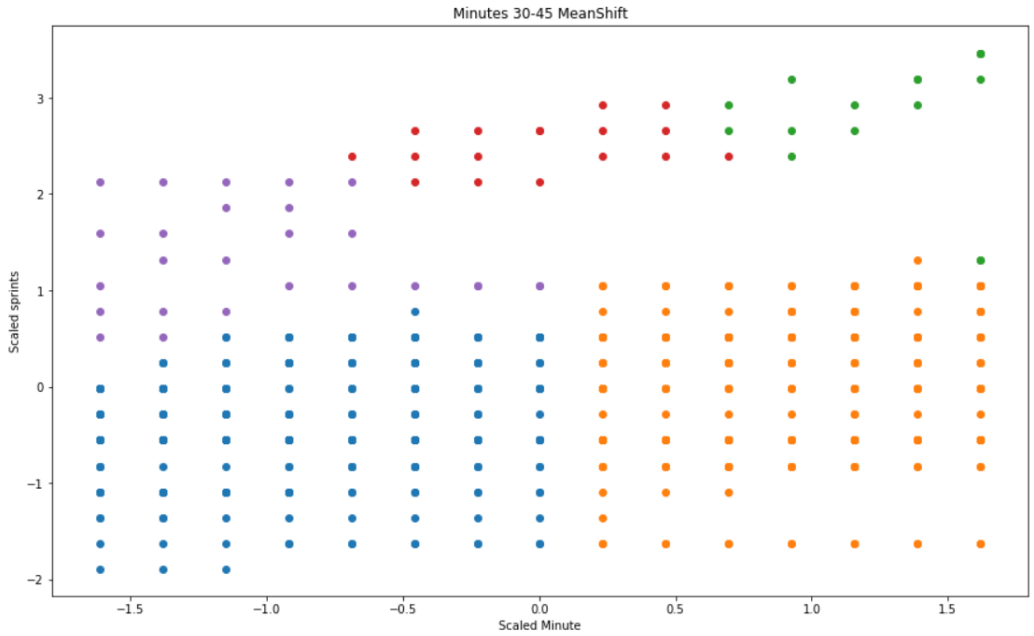

Abbildung 56: MeanShift Clustering des Intervalls 30-45 von *sprints, quantile = 0.15* (Eigendarstellung)

Bei anderen Intervallen, wie dem Intervall 75-90, wurden drei Bereiche sichtbar, doch diese wurden nicht gut horizontal voneinander getrennt und würden somit, später beim Klassifikator, zu falschen Klassifizierungen führen.

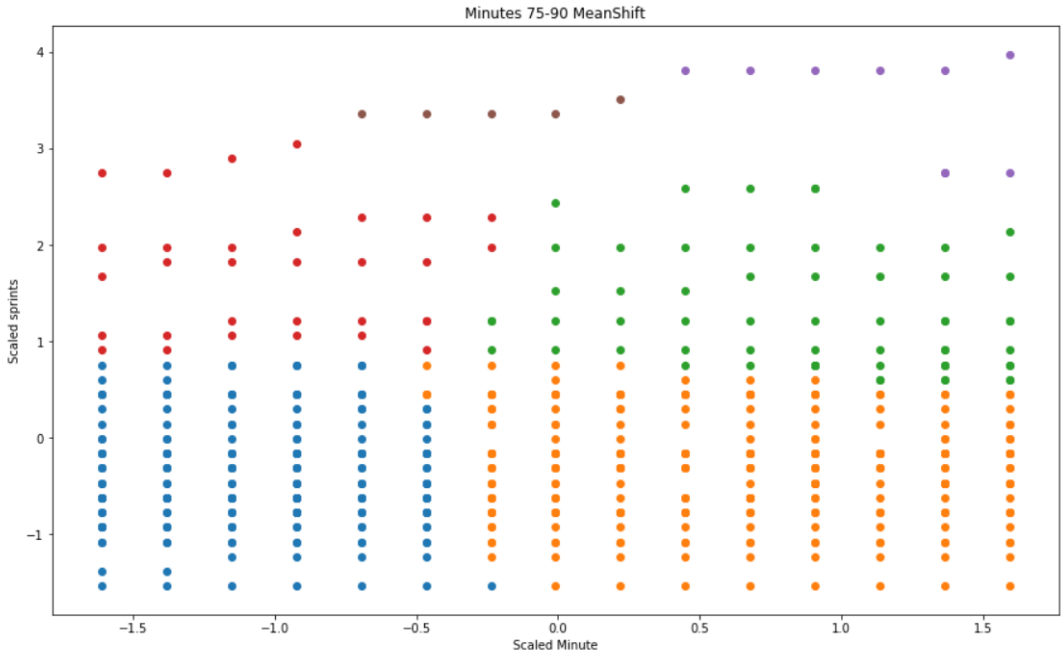

Abbildung 57: MeanShift Clustering des Intervalls 75-90 von *sprints*, *quantile = 0.15* (Eigendarstellung)

Daher wurde der Parameter *quantile* angepasst auf 0.06, dieser wurde verringert da er zur Berechnung der Bandbreite genutzt wird und es dadurch zu einer kleineren Bandbreite kommt, mit welcher kleinere Cluster erwartet werden. Diese könnten dann dafür sorgen, dass tatsächlich drei Bereiche gebildet werden und dass es zu genaueren Trennungen der Bereiche kommt. Dies funktionierte auch wie erwartet, es wurden deutlich mehr Cluster gebildet und alle Intervalle konnten nun in die drei gewünschten Bereiche unterteilt werden.

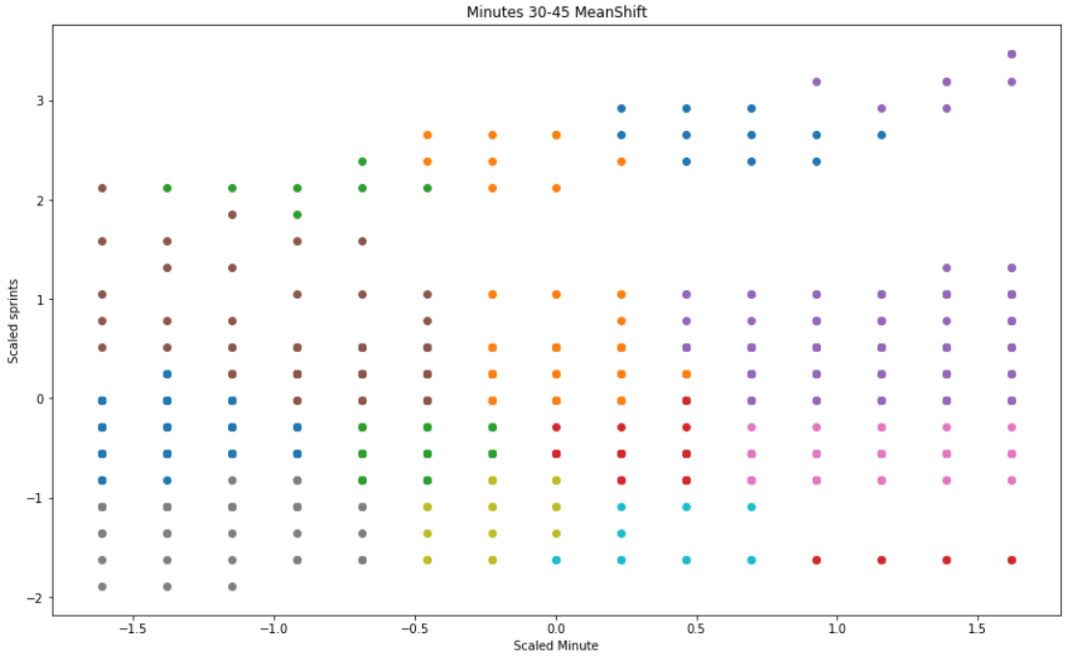

Abbildung 58: MeanShift Clustering des Intervalls 30-45 von *sprints, quantile = 0.06* (Eigendarstellung)

Doch nur weil eine Unterteilung in die drei Bereiche „Gut“, „Schlecht“ und „Durchschnittlich“ möglich ist, heißt dies nicht, dass das Clustering sinnvoll ist. Denn wie bei dem Intervall 30-45 zu sehen ist laufen die Cluster noch leicht in unterschiedliche Bereiche über. Die einzigen Intervalle die bei diesem Parameter gute Ergebnisse liefern sind die Intervalle 0-15 und 45-60, die restlichen sind zwar nicht schlecht, jedoch nicht ideal, sondern nur akzeptabel.

Bei dem Attribut *distanceCoveredNet* fallen die Ergebnisse mit diesem Parameter etwas besser aus. Der Großteil der Intervalle wird aufgrund der großen Anzahl der Cluster gut aufgeteilt und die Trennung der Bereiche verläuft hauptsächlich horizontal. Lediglich das Intervall 0-15 wird schlechter aufgeteilt, was, wie bei KMeans schon angemerkt, daran liegen könnte, dass dieses tatsächlich einen nicht behandelten Ausreißer beinhaltet.

Auch bei dem Attribut *intenseRunsNet* kommt es zu überwiegend guten Ergebnissen. Diese sind auf den ersten Blick nicht immer erkennbar, da aufgrund der hohen Anzahl an Clustern die Farben der Cluster für andere Cluster wieder verwendet werden.

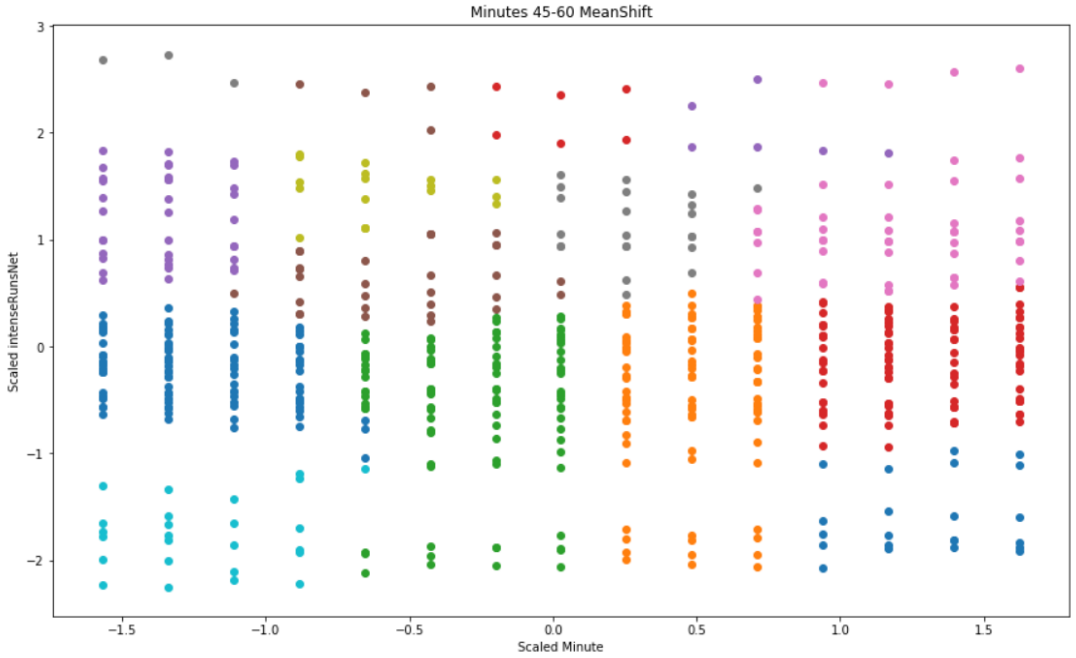

Abbildung 59: MeanShift Clustering des Intervalls 30-45 von *IntenseRunsNet*, *quantile = 0.06* (Eigendarstellung)

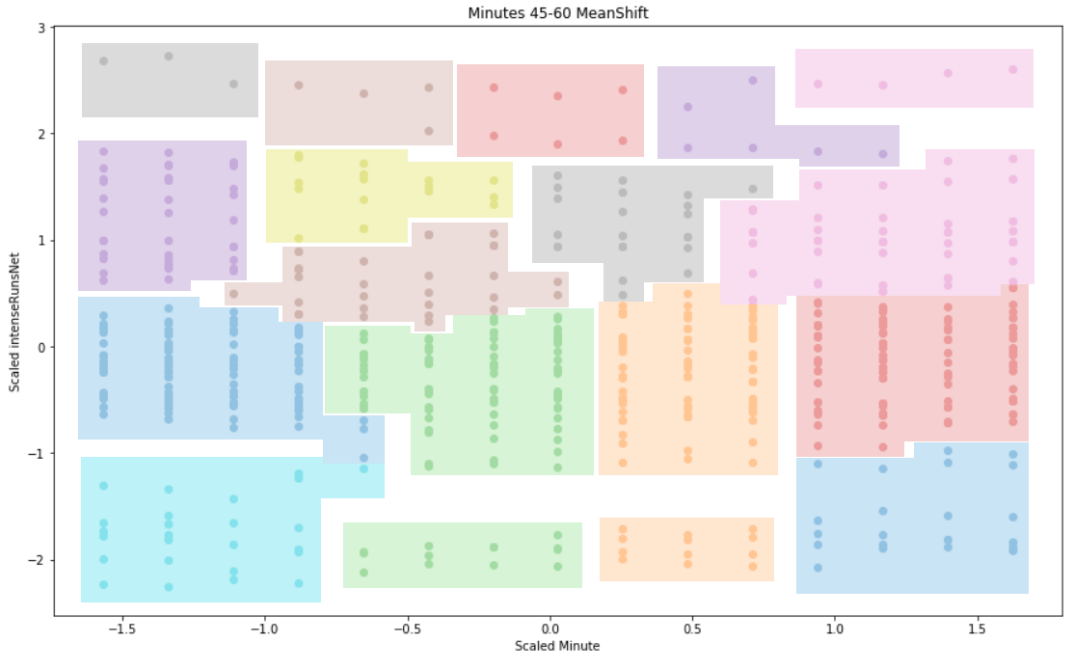

Abbildung 60: MeanShift Clustering des Intervalls 30-45 von *IntenseRunsNet*, *quantile = 0.06*. Verdeutlichte Anzeige der gebildeten Cluster (Eigendarstellung)

Mit MeanShift lassen sich mit einem Parameter für alle Attribute und Intervalle gute oder mindestens akzeptable Ergebnisse finden. Aufgrund der hohen Anzahl an gebildeten Clustern könnte es sein, dass dieses Verfahren bei dem Klassifikator noch genauer arbeitet als andere mit weniger Clustern, doch dies könnte auch mehr Zeit in Anspruch nehmen.

3.4 Diskussion

Die Verfahren bei welchem Parameter von Intervall zu Intervall oder von Attribut zu Attribut angepasst werden müssen, um gute Ergebnisse zu liefern, sind für den geplanten Klassifikator nicht geeignet. Denn dieser muss für alle Intervalle aller Attribute gute Ergebnisse liefern können und kann nicht zwischenzeitlich für ein bestimmtes Attribut oder ein bestimmtes Intervall angepasst werden. Daher entfallen die Verfahren BIRCH und DBSCAN für den Klassifikator. Von den übrigen Verfahren werden dann diejenigen gewählt, welche die besten Ergebnisse abgeliefert hatten.

Das Hierarchische Clustering bildet durch das Verbinden der am nächsten beieinander stehenden Werte oftmals Cluster, welche die gedachten und geplanten Grenzen der Bereiche „Gut“, „Schlecht“ und “Durchschnittlich“ überschreiten. Dementsprechend würde dieses Verfahren Werte oftmals falsch klassifizieren, weshalb es nicht im Klassifikator genutzt wird.

Das Spectral Clustering liefert zwar hauptsächlich akzeptable Ergebnisse, doch da die Cluster anhand der Verbundenheit ihrer Werte gebildet werden und nicht anhand der Kompaktness der Werte, kommt es auch hier stellenweise zu schlechten Ergebnissen. Diese würden zu falschen Klassifikationen führen, weshalb der Klassifikator nicht auf Spectral Clustering aufgebaut wird.

Affinity Propagation erzielt mit gleichbleibenden Parametern *damping* und *preference* größtenteils akzeptable Ergebnisse, aber auch hier kommt es zwischendurch zu schlechten Ergebnissen. Um zu guten Ergebnissen zu kommen, benötigt es eine, den Werten endsprechende, Auswahl der Exemplars und für die Berechnung dieser Exemplars sind diese Parameter *damping* und *preference* ausschlaggebend. Daher wird der Klassifikator nicht auf diesem Verfahren aufgebaut.

KMeans liefert mit acht Clustern, die um Zentren herum gebildet werden, größtenteils gute Ergebnisse über alle Intervalle und Attribute hinweg. Die schlechtesten Ergebnisse fallen hier immer noch akzeptabel aus. Daher ist KMeans ein Verfahren, mit welchem der Klassifikator gebaut werden kann.

MeanShift liefert ebenfalls über alle Intervalle und Attribute mit einem *quantile* von 0.06 gute oder im schlimmsten Fall akzeptable Ergebnisse. Hier fallen die Ergebnisse trotz gleichbleibenden Parameters gut aus, da dieser nicht direkt die Bandbreite des Kernels darstellt. Er geht stattdessen gemeinsam mit der Größe des Datensatzes in die Berechnung dieser ein. Aufgrund dieser guten Ergebnisse wird der Klassifikator auch auf Basis von MeanShift aufgebaut, somit können am Ende die Ergebnisse von MeanShift und KMeans verglichen werden.

4 Klassifikator

4.1 Implementierung

4.1.1 Basierend auf Standardabweichung

Bevor die Klassifikatoren basierend auf den Clustering-Verfahren gebaut wurden und bevor mit den Rohdaten aus den DFL-URLs gearbeitet wurde, entstand in der Wartezeit zwischen den verschiedenen Spieltagen ein Klassifikator, welcher die Werte anhand ihrer Standardabweichung klassifizierte. Dieser basierte auf den Daten der *dfldata* und nahm als Input einen Spielernamen, ein Attribut und eine Spielminute entgegen. Daraufhin wurde der Meanwert der, für den Spieler spezifischen, Attributs-Werte innerhalb des zur Minute gehörigen Intervalls berechnet und anhand dessen erfolgte die Berechnung der Standardabweichung. Alle Werte innerhalb der Standardabweichung um den Meanwert herum wurden als „Durchschnittlich“ klassifiziert, alle die darunter lagen als „Schlecht“ und alle die darüber lagen als „Gut“. Dieses Klassifikator-Programm gibt einen Graphen aus, welcher die Werte klassifiziert darstellt. Unter

5.1 sind die GUIs beschrieben, welche basierend auf diesem Programm aufgesetzt wurden. Das Programm enthält Methoden des Programmes *MDB_Data.py* welches unter 4.1.2 erläutert wird.

connectMongo.py

Programm, welches beim Start den Nutzer nach einem Spielernamen, einem Attribut und einer Spielminute abfragt und dann basierend darauf einen Graphen zurückgibt. In diesem sind die Werte des Spielers, die aus *MDB_Data.py* abgerufen werden, klassifiziert durch die Standardabweichung aufgezeigt. Der aktuelle Wert in *dfldata* des Spielers für das Attribut wird im Kontext der eingegebenen Spielminute ebenfalls klassifiziert. Dementsprechend ist in diesem Programm die klassifikations-Funktionalität enthalten. Es kann eigenständig ausgeführt werden, um zu Ergebnissen zu kommen und die klassifikations-Funktionalität wird von den GUIs in 6.1 zur Klassifikation genutzt.

Die GUIs nutzten die Methoden *connect(), pick_df(), get_value(), give_graph(), give_color_cluster() und get_defenders()* dieses Programmes, wobei indirekt auch die Hilfsmethoden *give_color_helper()*, *ret_color()*, *StandardAbweichung()* und *classifyVals()* genutzt werden.

connect(): Erstellt einen Mongo-Client und ruft auf diesem die gewünschte Datenbank *dfldata* auf. Die Collection *players* von dieser wird ausgewählt, welche die Spielerdaten beinhaltet. Rückgabewert ist die Collection an Spieler-Dictionarys *players*.

pick_df(name): Bekommt einen String *name* übergeben. Ruft basierend auf dem im String stehenden Namen die passende getter-Methode aus *MDB_Data.py* raus und gibt ein Tupel bestehend aus dem zum String *name* passenden Dataframe sowie dem Namen des Spielers als String zurück.

get_value(player_col, name, att): Bekommt eine Collection an Spielern *player_col*, einen String *name* und einen String *att* übergeben. Sucht in der Collection *player_col* nach dem Spielernamen *name* und sucht für diesen Spieler den Wert unter dem Attribut *att* raus. Gibt den Wert unter *att* des Spielers *name* zurück.

give_graph(df, attr, minu, curr_val, p_name): Bekommt einen Dataframe *df*, ein String *attr*, eine Zahl *minu*, ein Zahl *curr_val* und einen String *p_name* übergeben. Gibt einen Graphen

zurück, welcher die Werte aus dem Dataframe *df* des Spielers mit dem Namen *p_name* aus dem Minuten-Intervall der Minute *minu* beinhaltet. Dieser Graph zeigt den Wert *curr_val* als horizontale Linie an und klassifiziert den Wert in Bezug auf die Standardabweichung dieser Werte des Minuten-Intervalls.

get_defenders(): Gibt eine Liste mit Spielernamen als String zurück. In dieser stehen die Spieler des HSV, die auf der Position Verteidiger spielen und bei welchen genug Daten gesammelt wurden, um anhand dieser eine Klassifizierung zu ermöglichen.

standardAbweichung(values): Bekommt eine Liste *values* übergeben. Berechnet den Mean-wert von *values* und die Standardabweichung von *values* und gibt diese beiden Werte als Tupel zurück.

classifyVals(val, dev, mean): Bekommt drei Zahlen übergeben. Berechnet den Wert *plusDev* durch das Addieren der Werte *dev* und *mean* und den Wert *minusDev* durch das Subtrahieren des Wertes *dev* von dem Wert *mean*. Wenn sich *val* zwischen den Werten *plusDev* und *minusDev* befindet, wird eine null zurückgegeben. Wenn *val* über dem Wert *plusDev* liegt, wird eine eins zurückgegeben. Wenn *val* unter dem Wert *minusDev* liegt, wird eine minus eins zurückgegeben.

helper(player, val, minDF, minute): Bekommt den Name des Spielers *player* als String übergeben, sowie den zu klassifizierenden Wert *val* als Zahl. Außerdem bekommt die Methode noch einen Dataframe *minDF* und die Minute *minute* als Zahl übergeben. Ruft die Methode *standardAbweichung()* mit dem Parameter *minDF* und *values* auf und speichert den Rückgabewert in den Variablen *dev* und *mean*. Ruft dann die Methode *classifyVals()* mit den Parametern *val, dev* und *mean* auf und speichert den Rückgabewert in der Variablen *cluster*. Anhand von *cluster* wird der String *ret_string* bestimmt, welcher Auskunft darüber gibt in welchem Bereich sich *val* befindet, und zurückgegeben.

giveClusterHelper(player, attr, val, minute, theDF): Der Methode wird der Name des Spielers *player* sowie das Attribut *attr* als String übergeben. Außerdem werden der zu klassifizierender Wert *val* sowie die für den Bezug notwendige Spielminute *minute* als Zahl und der zum Spieler *player* gehörende Dataframe *theDF* übergeben (Dataframe Aufbau: *Zeit, Attribut1, Attribut2, …, Attribut9*). Zuerst bestimmt die Methode mit If-Abfragen in welchem Zeit-Intervall sich

minute befindet, und erstellt anhand dessen den angepassten Dataframe *df*. Danach wird die Methode *helper()* aufgerufen. An *helper()* werden übergeben: *player, val, df[attr]* und *minute*. Hierbei ist *df[attr]* ein Ausschnitt aus dem angepassten Dataframe *df*, dieser enthält jetzt nur die Zeit, die dem Intervall von *minute* zugeordnet ist und nur das Attribut *attr* (Aufbau von *df[attr]:* (*Zeit, Attribut*)).

giveCluster(playerName, attribute, value, minute): Der Methode werden der Name des Spielers *playerName* und das gewünschte Attribut *attribute* als Sting und der zu klassifizierende Wert *value* sowie die für den Bezug notwendige Spielminute *minute* als Zahl übergeben. Die Methode überprüft den entgegengenommenen Namen *playerName* und sucht den entsprechenden Dataframe zu diesem Namen aus. Dieser wird mithilfe der passenden getter-Methode von *MDB_Data.py* beschafft. Damit erfolgt dann der Aufruf der Methode *giveClusterHelper()*. Dieser Methode werden *playerName, attribute, value, minute* und der passende Dataframe *df* übergeben.

ret_color(val, df): Bekommt eine Zahl *val* sowie einen Dataframe *df* übergeben. Berechnet die Standardabweichung *dev* und den Meanwert des Dataframes *df* anhand der Methode s*tandardAbweichung()*. Speichert den Rückgabewert der Methode *classifyVals()* in der Variablen *cluster*. Gibt Anhand des Wertes in *cluster* einen Hexadezimalcode als String zurück.

give_color_helper(attr, val, minute, curr_df): Bekommt den Sting *attr*, die Zahlen *val* und *minute* sowie den Dataframe *curr_df* übergeben. Sucht Anhand von *minute* das korrekte Minuten-Intervall heraus und beschränkt den Dataframe *curr_df* auf dieses Intervall. Rückgabewert ist der Rückgabewert der Methode *ret_color()*.

give_color_cluster(player_name, attribute, minute, value): Bekommt die Strings *player_name* und *attribute* sowie die Zahlen *minute* und *value* übergeben. Sucht Anhand von *player_name* die korrekte getter-Methode aus *MDB_Data.py* heraus. Rückgabewert ist der Rückgabewert der Methode *give_color _helper()*.

4.1.2 Basierend auf Clustering

Die Implementierungen der Klassifikatoren für die Daten aus der Datenbank *dfldata* und die Rohdaten aus den DFL-URLs ähneln sich. Der Unterschied zwischen den beiden stellt sich

durch die zugrundeliegenden Daten und durch die verfügbaren Clustering-Verfahren dar. Die Unterschiede im Code beziehen sich auf die CSV-Dateien, aus welchen die Daten bestehen. Bei diesen wird die Spielzeit in den Rohdaten der DFL-URLs unter dem Namen „Minute“ abgespeichert, während sie in den Daten der Datenbank *dfldata* unter dem Namen „Zeit“ abgespeichert werden. Der Unterschied der Clustering-Verfahren bezieht sich darauf, dass bei den DFL-URL Rohdaten eine Auswahl zwischen MeanShift und KMeans erfolgen kann, während bei den Daten der *dfldata* nur KMeans verfügbar ist. Beide Programme haben ihr jeweiliges Hilfsprogramm, *MDB_Data.py* oder *DFL_Data.py*, welches die zugehörigen Daten zur Verfügung stellt. Hierbei ermöglicht *MDB_Data.py* pro Spieler einen, zu ihm gehörigen, Dataframe abzurufen, auf dessen Basis die Clusterung und Klassifikation dann erfolgt. Im Gegensatz dazu kann bei *DFL_Data.py* nur ein Dataframe abgerufen werden, welcher die Werte aller Spieler beinhaltet, auf dessen Basis bei allen Spielern die Clusterung und Klassifikation erfolgt.

Beim Start der Programme werden diesen der Name des Attributs übergeben, dessen Wert klassifiziert werden soll, der Wert dieses Attributes und die Spielminute, zu welcher der Wert in Kontext gesetzt werden soll und ein String zur Wahl des Klassifikators. Bei dem Programm der DFL-URL Daten wird des Weiteren ein String zur Wahl des Clustering-Verfahrens übergeben. Zusätzlich wird eine boolesche Variable übergeben, welcher wenn sie als *False* übergeben wurde, den Print von Ausgaben und Graphen verhindert. Die Programme geben, wenn die boolesche Variable als *True* übergeben wurde, über eine Print-Ausgabe heraus, ob der Wert in dieser Minute als gut, schlecht oder durchschnittlich klassifiziert wurde. Des Weiteren werden zwei Graphen ausgegeben, der erste plottet die Werte dieses Attributes in der zugehörigen Spielminute, der zweite zeigt die Ergebnisse der Clusterung an.

Die Klassifikatoren werden hierbei darauf trainiert, den übergebenen Wert einem der vorher erstellten Cluster zuzuordnen. Bei beiden werden diese Cluster anhand des Meanwertes aller in ihnen enthaltenen Werte einem Bereich „Gut“, „Schlecht“ oder „Durchschnittlich“ zugeordnet. Diese Bereiche werden durch vorher Berechnete Schwellenwerte getrennt. Die Schwellenwerte werden berechnet, indem das Werteintervall des Attributs in jenem Minuten Intervall durch Drei geteilt wird und dann dieser Wert jeweils einmal zum Werte Minimum hinzugefügt und vom Werte Maximum abgezogen wird. Somit wird umgangen immer eine feste Anzahl an Clustern einem Bereich zuweisen zu müssen.

Das Programm der *dfldata* Daten ist *classifier_MDB.py* und das Programm für die Rohdaten der DFL-URLs ist *classifier_DFL.py*.

MDB_Data.py

Liest die CSV-Dateien, der aufgezeichneten HSV-Spieler aus *dfldata*, ein und erstellt jeweils für jeden Spieler ein Dataframe sowie ein Dataframe, welcher die Werte alles Spieler enthält. Enthält fünf getter-Methoden, die jeweils den Dataframe von einem von fünf Spielern zurückgeben und eine getter-Methode, welche den gemeinsamen Dataframe aller Spieler zurückgibt.

DFL_Data.py

Liest die CSV-Dateien aller aus den DFL-URLs aufgezeichneten Spieler ein und erstellt einen Dataframe, welcher die Werte aller Spieler enthält. Enthält eine getter-Methode, welche den gemeinsamen Dataframe aller Spieler zurückgibt.

get_data(): Gibt den Dataframe aller aufgezeichneten Spieler zurück.

classifier_MDB.py

Programm, welches einen Wert, eine Minute und ein Attribut für einen Spieler auf Basis der KMeans Clusterung mit sieben Clustern klassifiziert. Die sieben erstellten Cluster werden in die Bereiche „Gut“, „Schlecht“ und „Durchschnittlich“ anhand ihrer Meanwerte und zwei berechneter Schwellenwerte unterteilt. Die Cluster, deren Meanwerte über dem zweiten Schwellenwert liegen, werden dem Bereich „Gut“ zugeordnet, die Cluster, deren Meanwerte unter dem ersten Schwellenwert liegen, dem Bereich „Schlecht“ und die Cluster, deren Meanwerte zwischen den beiden Schwellenwerten liegen, werden dem Bereich „Durchschnittlich“ zugeordnet. Ermöglicht die Auswahl zwischen den Klassifikatoren Naive Bayes, Decision Tree und K-Nearest-Neighbor. Gibt als Rückgabewert, abhängig von dem Bereich, welchem der Wert zugeordnet wurde, eine Zahl zurück.

choose_df(name): Bekommt einen Spielernamen als String übergeben. Ruft entsprechend dem Namen im String die getter-Methode von *MDB_Data.py* auf. Gibt als Rückgabewert die dem Spieler zugehörigen Dataframe zurück.

cut_df(df, minute, att): Bekommt einen Dataframe *df*, eine Zahl *minute* und ein String *att* übergeben. Wählt anhand von *minute* das korrekte Minuten-Intervall. Gibt als Rückgabewert den Ausschnitt des Dataframes *df*, bei welchem die Werte der Spalte „Zeit“ im Minuten-Intervall enthalten sind.

scale_val(df): Bekommt ein Dataframe *df* übergeben. Skaliert den Dataframe mit dem *StandardScaler* und gibt die letzte Zeile des Dataframes als *numpy.ndarray* zurück.

context_val(df, val, att, out): Bekommt einen Dataframe *df*, eine Liste *val*, einen String *att* und eine boolesche Variable *out* übergeben. Falls die Variable *out* als *True* übergeben wird, dann werden Prints und Graphen ausgegeben. Die Liste *val* besteht aus zwei Zahlen, von welchen die erste die Spielminute repräsentiert und die zweite einen Wert eines Attributes. Fügt dem Dataframe *df* die Liste *val* als weitere Zeile hinzu und übergibt diesen neuen Dataframe an die Methode *scale_val()*. Gibt als Rückgabewert den Rückgabewert von *scale_val()* zurück.

scale_df(df, att): Bekommt einen Dataframe *df* und ein Attribut *att* als String übergeben. Skaliert den Dataframe *df* mit dem *StandardScaler* und gibt den skalierten Dataframe zurück.

classify (df, att, classifier, t1, t2, out): Bekommt einen Dataframe *df*, ein Attribut *att* als String, zwei Zahlen *t1* und *t2*, einen String *classifier* sowie eine boolesche Variable *out* übergeben. Falls *out* als *True* übergeben wird, dann werden Prints und Graphen ausgegeben. Clustert die Werte aus *df* mit KMeans und sieben Clustern und fügt *df* eine Spalte *Cluster* hinzu, in welcher steht in welchem Cluster sich die jeweilige Zeile befindet. Dieser bearbeitete Dataframe wird als *temp* gespeichert. Teilt den Dataframe *temp* in Trainings- und Testdaten ein. Erstellt abhängig davon, was in *classifier* steht entweder einen Naive Bayes Klassifikator, ein Decision Tree Klassifikator oder einen K-Nearest-Neighbor Klassifikator. Fittet den erstellten Klassifikator *model* mit den Trainingsdaten und gibt *model*, den Dataframe *temp* und eine Liste der vorhandenen Cluster *clusters* zurück.

good_bad(df, pred, att, c_lst, t1, t2, out): Bekommt einen Dataframe *df*, eine Liste *pred*, einen String *att*, eine Liste *c_lst*, zwei Zahlen *t1* und *t2* und eine boolesche Variable *out* übergeben. Falls *out* als *True* übergeben wird, dann werden Prints und Graphen ausgegeben. Erstellt ein Dictionary *mean_dict* in welchem jeweils die Nummer der Cluster als Keys und der Meanwert der in dem Cluster enthaltenen Werte als Value gespeichert wird. Erstellt außerdem drei Listen

goods, *bads* und *avers*. Fügt alle Cluster, deren Meanwert unter *t1* liegt der Liste *bads* hinzu, alle Cluster deren Meanwert über *t2* liegt der Liste *goods* und alle Cluster deren Meanwert zwischen *t1* und *t2* liegt der Liste *avers* hinzu. Gibt abhängig davon in welcher Liste (*goods*, *bads* oder *avers*) sich der Wert des Arrays *pred* befindet eine Zahl zurück. Befindet er sich in der Liste *goods* wird eine eins zurückgegeben, befindet er sich in der Liste *bads* wird eine minus eins zurückgegeben und befindet er sich in der Liste *avers* wird eine null zurückgegeben.

give_class(name, val, att, classifier, out): Bekommt einen String *name*, eine Liste *val*, ein Attribut als String *att*, einen String *classifier* und einen booleschen Wert *out* übergeben. Falls die Variable *out* als *True* übergeben wird, dann werden Prints und Graphen ausgegeben. Die Liste *val* besteht aus zwei Zahlen, von welchen die erste die Spielminute repräsentiert und die zweite einen Wert eines Attributes. Speichert sich die Minute *minute* aus der Liste *val*. Sucht sich mit der Methode *choose_df()* den zu *name* passenden Dataframe *df* raus. Schneidet mit der Methode *cut_df()* alle Werte aus *df* heraus, die sich nicht in dem Minuten-Intervall von *minute* befinden. Skaliert die Werte in der Liste *val* mit der Methode *scale_val()* und speichert dieses Ergebnis als *scaled_val* ab, skaliert außerdem *df* mit der Methode *scale_df()*. Errechnet die Variablen *t1* und *t2* mithilfe des Maximums und Minimums der Werte von *att* in *scaled_df*. Erstellt mit der Methode *classify()* den Klassifikator *model*, den erweiterten Dataframe *cluster_df* und eine Liste mit Clustern *clusters*. Erstellt mit *model* eine Vorhersage darüber, in welchem Cluster sich *scaled_val* befindet und speichert diese Vorhersage in *pred*. Ermittelt mit der Methode *good_bad()*, ob *pred* dem Bereich „Gut“, „Schlecht“ oder „Durchschnittlich“ zugeordnet ist. Gibt als Rückgabewert den Rückgabewert von *good_bad()* zurück.

classifier_DFL.py

Programm, welches einen Wert, eine Minute und ein Attribut auf Basis von KMeans mit acht Clustern oder MeanShift mit einem *quantile* Wert von 0.06 klassifiziert. Die erstellten Cluster werden in die Bereiche „Gut“, „Schlecht“ und „Durchschnittlich“ anhand von zwei berechneten Schwellenwerten aufgeteilt. Die Cluster, deren Meanwerte über dem zweiten Schwellenwert liegen, werden dem Bereich „Gut“ zugeordnet, die Cluster, deren Meanwerte unter dem ersten Schwellenwert liegen, dem Bereich „Schlecht“ und die Cluster, deren Meanwerte zwischen den beiden Schwellenwerten liegen, werden dem Bereich „Durchschnittlich“ zugeordnet. Ermöglicht die Auswahl zwischen den Klassifikatoren Naive Bayes, Decision Tree und K-

Nearest-Neighbor. Gibt als Rückgabewert, abhängig von dem Bereich, welchem der Wert zugeordnet wurde, eine Zahl zurück.

cut(minute, df, att): Bekommt eine Zahl *minute*, einen Dataframe *df* und ein String *att* übergeben. Schneidet aus *df* das zu *minute* zugehörige Minuten-Intervall heraus und gibt diesen neuen Dataframe zurück.

df_cut(df, minu, att): Bekommt einen Dataframe *df*, ein String *att* und eine Zahl *minu* übergeben. Anhand von *minu* wird entschieden welche Zahl der Methode *cut()* zusammen mit *df* und *att* übergeben wird. Rückgabewert ist der Rückgabewert von *cut()*.

scale_val(df): Bekommt ein Dataframe *df* übergeben. Skaliert den Dataframe mit dem *StandardScaler* und gibt die letzte Zeile des Dataframes als *numpy.ndarray* zurück.

context_val(df, val, att): Bekommt einen Dataframe *df*, eine Liste *val* und einen String *att* übergeben. Die Liste *val* besteht aus zwei Zahlen, von welchen die erste die Spielminute repräsentiert und die zweite einen Wert eines Attributes. Fügt dem Dataframe *df* die Liste *val* als weitere Zeile hinzu und übergibt diesen neuen Dataframe an die Methode *scale_val()*. Gibt als Rückgabewert den Rückgabewert von *scale_val()* zurück.

scale_df(df, att): Bekommt einen Dataframe *df* und ein Attribut *att* als String übergeben. Skaliert den Dataframe *df* mit dem *StandardScaler* und gibt den skalierten Dataframe zurück.

classify_MeanShift(df, att, t1, t2, classifier, out): Bekommt einen Dataframe *df*, ein Attribut *att* als String, zwei Zahlen *t1* und *t2*, einen String *classifier* sowie eine boolesche Variable *out* übergeben. Falls *out* als *True* übergeben wird, dann werden Prints und Graphen ausgegeben. Clustert die Werte aus *df* mit MeanShift und erstellt fügt *df* eine Spalte *Cluster* hinzu, in welcher steht in welchem Cluster sich die jeweilige Zeile befindet. Dieser bearbeitete Dataframe wird als *temp* gespeichert. Teilt den Dataframe *temp* in Trainings- und Testdaten ein. Erstellt abhängig davon, was in *classifier* steht entweder einen Naive Bayes Klassifikator, ein Decision Tree Klassifikator oder einen K-Nearest-Neighbor Klassifikator. Fittet den erstellten Klassifikator *model* mit den Trainingsdaten und gibt *model*, den Dataframe *temp* und eine Liste der vorhandenen Cluster *clusters* zurück.

classify_KMeans(df, att, t1, t2, classifier, out): Bekommt einen Dataframe *df*, ein Attribut *att* als String, zwei Zahlen *t1* und *t2*, einen String *classifier* sowie eine boolesche Variable *out* übergeben. Falls *out* als *True* übergeben wird, dann werden Prints und Graphen ausgegeben. Clustert die Werte aus *df* mit KMeans und erstellt fügt *df* eine Spalte *Cluster* hinzu, in welcher steht in welchem Cluster sich die jeweilige Zeile befindet. Dieser bearbeitete Dataframe wird als *temp* gespeichert. Teilt den Dataframe *temp* in Trainings- und Testdaten ein. Erstellt abhängig davon, was in *classifier* steht entweder einen Naive Bayes Klassifikator, ein Decision Tree Klassifikator oder einen K-Nearest-Neighbor Klassifikator. Fittet den erstellten Klassifikator *model* mit den Trainingsdaten und gibt *model*, den Dataframe *temp* und eine Liste der vorhandenen Cluster *clusters* zurück.

good_bad(df, pred, att, c_lst, t1, t2, out): Bekommt einen Dataframe *df*, eine Liste *pred*, einen String *att*, eine Liste *c_lst*, zwei Zahlen *t1* und *t2* und eine boolesche Variable *out* übergeben. Falls *out* als *True* übergeben wird, dann werden Prints und Graphen ausgegeben. Erstellt ein Dictionary *mean_dict* in welchem jeweils die Nummer der Cluster als Keys und der Meanwert der in dem Cluster enthaltenen Werte als Value gespeichert wird. Erstellt außerdem drei Listen *goods*, *bads* und *avers*. Fügt alle Cluster, deren Meanwert unter *t1* liegt der Liste *bads* hinzu, alle Cluster deren Meanwert über *t2* liegt der Liste *goods* und alle Cluster deren Meanwert zwischen *t1* und *t2* liegt der Liste *avers* hinzu. Gibt abhängig davon in welcher Liste (*goods*, *bads* oder *avers*) sich der Wert des Arrays *pred* befindet eine Zahl zurück. Befindet er sich in der Liste *goods* wird eine eins zurückgegeben, befindet er sich in der Liste *bads* wird eine minus eins zurückgegeben und befindet er sich in der Liste *avers* wird eine null zurückgegeben.

classify_with_threshold(att, val, name, classifier, out): Bekommt drei Strings *att, name* und *classifier*, eine Liste *val* und eine boolesche Variable *out* übergeben. Falls *out* als *True* übergeben wird, dann werden Prints und Graphen ausgegeben. Speichert in der Variablen *df* den Rückgabewert aus *DFL_Data.get_data()* und in der Variablen *minute* den ersten Wert der Liste *val*. Ruft die Methode *df_cut()* mit den Parametern *df, minute* und *att* auf und speichert den Rückgabewert in der Variablen *new_df*. Ruft die Methode *context_val()* mit den Parametern *new_df, val* und *att* auf und speichert den Rückgabewert in der Variablen *scaled_val*. Ruft die Methode *scale_df()* mit den Parametern *new_df* und *att* auf und speichert den Rückgabewert in der Variablen *scaled_df*. Errechnet die Variablen *t1* und *t2* mithilfe des Maximums und

Minimums der Werte von *att* in *scaled_df*. Ruft, abhängig von *name*, entweder die Methode *classify_MeanShift()* oder *classifiy_KMeans()* mit den Parametern *scaled_df, att, t1, t2, classifier* und *out* auf und speichert den Rückgabewert in den Variablen *model, cluster_df* und *cluster_lst*. Erstellt anhand von *model* eine Vorhersage darüber, in welchem Cluster sich der Wert *scaled_val* befinden würde und speichert diese Vorhersage unter der Variablen *prediction*. Ruft schlussendlich die Methode *good_bad()* mit den Parametern *cluster_df, prediction, att, cluster_lst, t1, t2* und *out* auf und speichert den Rückgabewert in der Variablen *klasse*. Gibt *klasse* zurück.

4.2 Ergebnisse

Der Klassifikator, welcher auf Basis der Standardabweichung aufgebaut wurde, wurde in diesem Abschnitt nicht berücksichtigt, da es sich bei diesem um keinen trainierbaren Klassifikator handelt.

4.2.1 *dfldata*

Es werden im Folgenden nur die Klassifikatoren der auf *dfldata* basierenden Daten und der auf den DFL-URLs basierenden Rohdaten verglichen. Dabei werden die Ergebnisse nicht von Datenquelle zu Datenquelle verglichen, sondern nur innerhalb der jeweiligen Datenquelle. Die durchgeführten Tests von *dfldata* unterscheiden sich etwas von jenen die bei den Rohdaten der DFL-URLs durchgeführt wurden. Bei den Tests der *dfldata* werden die drei Klassifikatoren anhand von vier Attributen getestet. Die Daten, auf welchen die Tests durchgeführt wurden, beinhalten Spiele der Spieltage 24 bis 29 und bestehen aus insgesamt 3079 Zeilen. Diese wurden in sechs 15 Minuten Intervall Dataframes aufgeteilt und mit dem *StandardScaler* skaliert. Jedem Test wurde ein spielerspezifischer Dataframe sowie ein Attribut und der gewünschte Klassifikator übergeben. Die Tests wurden pro Attribut auf einem der sechs 15 Minuten-Intervalle für jeden der fünf Verteidigern 167-mal durchgeführt. Damit ergeben sich pro Klassifikator und Attribut 835 Durchläufe, bei welchen jeweils die Genauigkeit des Klassifikators ermittelt wird. Diese Genauigkeit wird dann einer, für jeden Klassifikator neu initialisierten, Liste hinzugefügt. Am Ende wird dann der Meanwert der Liste berechnet, indem die Summe der in der Liste enthaltenen Genauigkeiten durch die Länge der Liste geteilt wird. Pro Attribut und

Minuten-Intervall werden die Ergebnisse dann ausgegeben. Bei allen Tests wird auf print-Ausgaben sowie Ausgaben von Graphen verzichtet.

Tabelle 1: Genauigkeit der Klassifikatoren bei den *dfldata* Daten

	sprints	**distance**	**averageSpeed**	**shotsAtGoal**
Naive Bayes	92.00%	90.45%	91.62%	96.18%
Decision Tree	92.27%	91.15%	91.23%	98.11%
K-Nearest-Neighbor	92.79%	93.32%	93.86%	95.11%

Der K-Nearest-Neighbor Klassifikator schneidet in drei Attributen am besten ab, nur im Attribut *shotsAtGoal* ist der Decision Tree besser. Der Naive Bayes Klassifikator konnte in keinem Attribut die besten Ergebnisse erzielen. Alle Klassifikatoren leiden an Overfitting, dabei ist der Decision Tree am stärksten betroffen, gefolgt vom Naive Bayes Klassifikator.

4.2.2 DFL-URL Rohdaten

Um die Klassifikatoren und die Clustering-Verfahren der DFL-URL Rohdaten zu vergleichen, wurden die folgende Tests durchgeführt. Für jedes Clustering-Verfahren wurden alle drei Klassifikatoren anhand von vier Attributen getestet. Die Daten, auf welchen die Tests durchgeführt wurden, beinhalten Spiele der Spieltage 28, 29, 30 und 32 (Genauer genommen die Spiele: Greuther Führt vs. Sandhausen, Kiel vs. Nürnberg, HSV vs. Karlsruhe, Kiel vs. Sandhausen, Kiel vs. St. Pauli und Hannover vs. Darmstadt). Diese wurden mit dem *StandardScaler* skaliert und in sechs 15 Minuten Intervall Dataframes aufgeteilt, welche zwischen 421 und 605 Zeilen beinhalteten. Jedem Test wurden ein Attribut übergeben sowie ein 15 Minuten-Intervall-Dataframe. Die Tests wurden pro Attribut auf allen sechs 15 Minuten-Intervallen jeweils 167-mal durchgeführt. Damit ergeben sich pro Klassifikator und Clustering-Verfahren 996 Durchläufe, bei welchen jeweils die Genauigkeit des Klassifikators ermittelt wird. Diese Genauigkeit wird dann einer, für jeden Klassifikator neu initialisierten, Liste hinzugefügt. Am Ende wird dann der Meanwert der Liste berechnet, indem die Summe der in der Liste enthaltenen

Genauigkeiten durch die Länge der Liste geteilt wird. Pro Attribut werden die Ergebnisse dann ausgegeben. Bei allen Tests wird auf print-Ausgaben sowie Ausgaben von Graphen verzichtet.

Naive Bayes

Tabelle 2: Naive Bayes Genauigkeit der Clustering-Verfahren bei den DFL-URL Rohdaten

	sprints	**intenseRunsNet**	**distanceCoverednet**	**averageSpeed**
KMeans	95.60%	97.18%	97.27%	96.31%
MeanShift	93.02%	91.74%	92.90%	93.43%

Bei den DFL-URL Rohdaten schneidet der Naive Bayes Klassifikator bei KMeans in allen getesteten Attributen besser ab als bei MeanShift. Die Genauigkeit von KMeans mit dem Naive Bayes Klassifikator lag am höchsten bei dem Attribut *distanceCoveredNet*, während MeanShift seine höchste Genauigkeit beim Attribut *averageSpeed* hatte.

Decision Tree

Tabelle 3: Decision Tree Genauigkeit der Clustering-Verfahren bei den DFL-URL Rohdaten

	sprints	**intenseRunsNet**	**distanceCoverednet**	**averageSpeed**
KMeans	99.04%	96.58%	97.75%	96.01%
MeanShift	96.65%	93.30%	93.70%	94.08%

Bei den DFL-URL Rohdaten schneidet der Decision Tree als Klassifikator bei KMeans erneut in allen getesteten Attributen besser ab als bei MeanShift. Im Vergleich zum Naive Bayes Klassifikator ist die Genauigkeit von MeanShift in jedem Attribut besser mit dem Decision Tree. KMeans verbessert sich im Vergleich zum Naive Bayes Klassifikator in den Attributen *sprints* und *distanceCoveredNet*.

K-Nearest-neighbor

Tabelle 4: K-Nearest-Neighbor Genauigkeit der Clustering-Verfahren bei den DFL-URL Rohdaten

	sprints	**intenseRunsNet**	**distanceCoverednet**	**averageSpeed**
KMeans	97.40%	96.08%	96.80%	96.09%
MeanShift	94.40%	90.96%	92.94%	93.70%

Wie auch bei den vorherigen Ergebnissen der DFL-URL Rohdaten liegt die Genauigkeit von KMeans in jedem getesteten Attribut über der von MeanShift. Im Vergleich zum Decision Tree Klassifikator ist KMeans nur in dem Attribut *averageSpeed* leicht besser als dieser und im Vergleich zum Naive Bayes Klassifikator ist KMeans nur im Attribut *sprints* besser als jener. MeanShift ist im Vergleich zu seinen Ergebnissen aus dem Decision Tree Klassifikator in jedem Attribut schlechter, im Vergleich zu den Ergebnissen des Naive Bayes verbessert sich die Genauigkeit in jedem Attribut außer *intenseRunsNet*.

4.3 Diskussion

Bei den Daten der *dfldata* schneiden der K-Nearest-Neighbor am besten ab, gefolgt vom Decision Tree. Der Naive Bayes Klassifikator schneidet am schlechtesten von den drei ab. Bei den Rohdaten der DFL-URLs schneidet der Decision Tree insgesamt bei beiden Clustering-Verfahren am besten ab. Alle Klassifikatoren beider Datenquellen leiden aufgrund der geringen Menge von Daten am Overfitting, dabei ist bei beiden der Decision Tree am stärksten betroffen. Dies kann mit der Verringerung der maximalen Tiefe des gebildeten Baumes *max_depth* behoben werden. Allerdings leidet dann die Genauigkeit des Klassifikators darunter.

Tabelle 5: Trainings Genauigkeit des Decision Trees bei den *dfldata* Daten

Training Accuracy	**sprints**	**distance**	**averageSpeed**	**shotsAtGoal**
Unbegrenzte Tiefe	100.0%	100.0%	100.0%	100.0%
Maximale Tiefe = 4	95.47%	95.96%	96.40%	93.03%

Tabelle 6: Genauigkeit des Decision Trees bei den *dfldata* Daten

Testing Accuracy	**sprints**	**distance**	**averageSpeed**	**shotsAtGoal**
Unbegrenzte Tiefe	92.27%	91.15%	91.23%	98.11%
Maximale Tiefe = 4	87.52%	87.31%	86.92%	83.83%

Es werden daher bei beiden Datenquellen größere Datensätze benötigt, um Aussagekräftigere Ergebnisse erzielen zu können.

Auffällig bei den Tests der DFL-Daten ist, dass die Tests mit KMeans als Clustering-Verfahren deutlich schneller durchlaufen als die mit MeansShift. So braucht einer der, oben beschriebenen, Test für das Attribut *distanceCoveredNet* bei KMeans 02:55 Minuten, um durchzulaufen und die Ergebnisse zu liefern. Bei MeanShift benötigt derselbe Test 01:45:06 Stunden. Schon beim Clustering fiel auf, dass MeanShift länger braucht, um zu Ergebnissen zu kommen als KMeans. Dies liegt wahrscheinlich daran, dass MeanShift eine höhere Anzahl an Clustern bildet als KMeans. Da jedoch KMeans in allen getesteten Attributen und Klassifikatoren besser abschneidet als MeanShift, ist es in Betracht zu ziehen, MeanShift aufgrund der längeren Durchlaufzeit als Clustering-Verfahren zu verwerfen.

5 Visualisierung

Die Visualisierung Erfolgte in zwei Unterschiedlichen GUIs, wobei eine nur auf Basis der Standardabweichung aufgebaut ist. Die zweite GUI ist diejenige, welche die Clustering-Verfahren und die Klassifikatoren miteinbezieht. Die GUIs, welche mit den Daten der *dfldata* arbeiten, beziehen ihre Daten aus dem Programm *MDB_Data.py,* dieses beinhaltet Spiele der

Spieltage 24-29. GUIs, welche mit Rohdaten aus den DFL-URLs arbeiten, beziehen ihre Daten aus dem Programm *DFL_Data.py*, dieses beinhaltet Spiele der Spieltage 28-30 und 32.

5.1 GUI Standardabweichung

Es wurden zwei GUIs basierend auf der Standardabweichung gebaut. Beide wurden mit Hilfe von TKinter erstellt und beide beziehen ihre Daten aus *MDB_Data.py*.

5.1.1 GUIs

GUI_Standardabweichung.py

Die erstellte GUI wurde anhand des Standardabweichung-Klassifikators gebaut. Diese GUI ist nur auf Spieler des HSV ausgelegt und beinhaltet nur Verteidiger des HSV, die schon mindestens fünf Spiele als Verteidiger absolviert haben.

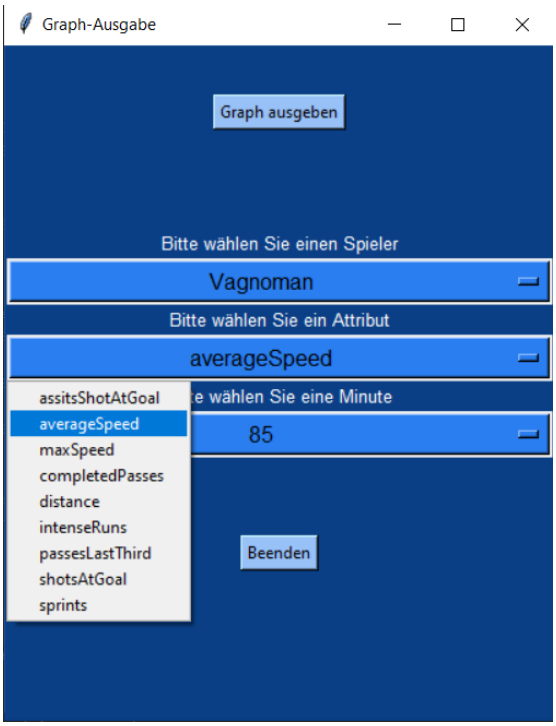

Abbildung 61: Initialer Screen der GUI *GUI_Standardabweichung.py* (Eigendarstellung)

Sie wurde mit TKinter erstellt und sie ermöglicht dem Nutzer einen von vier vordefinierten Spielern sowie eins von neun Attributen und eine Minute von 1-90 auszuwählen. Diese Möglichkeiten sind durch Option-Menus wählbar. Im oberen Bereich der GUI befindet sich ein Button mit der Aufschrift „Graph ausgeben“ und in dem unteren Bereich ein weiterer Button mit der Aufschrift „Beenden“. Der „Beenden“-Button schließt die GUI. Der „Graph ausgeben“-Button fragt den aktuellen Wert *W* des ausgewählten Attributs und des ausgewählten Spielers in der Datenbank *dfldata* ab. Der von der Datenbank erfragte Wert *W* wird dann in Kontext zur ausgewählten Spielminute gesetzt und innerhalb eines Graphen anhand der Standardabweichung klassifiziert zurückgegeben. Für die Berechnung der Standardabweichung greift die GUI auf vordefinierte Dataframes zu, wobei jedem Spieler ein ihm zugehöriger Dataframe *D* zur Verfügung steht.

Der ausgegebene Graph zeigt auf der X-Achse die Minuten an und auf der Y-Achse den Wertebereich des ausgewählten Attributes des Spielers. Außerdem werden im Graphen die Werte aus dem Dataframe *D* aufgezeigt, welche in dem zugehörigen Minuten Intervall enthalten sind und welche zur Berechnung der Standardabweichung genutzt wurden. Die schon vorhandenen Werte unterscheiden sich farblich, je nachdem ob sie über, unter oder innerhalb der Standardabweichung liegen. Der aktuelle Wert wird als horizontale Linie dargestellt.

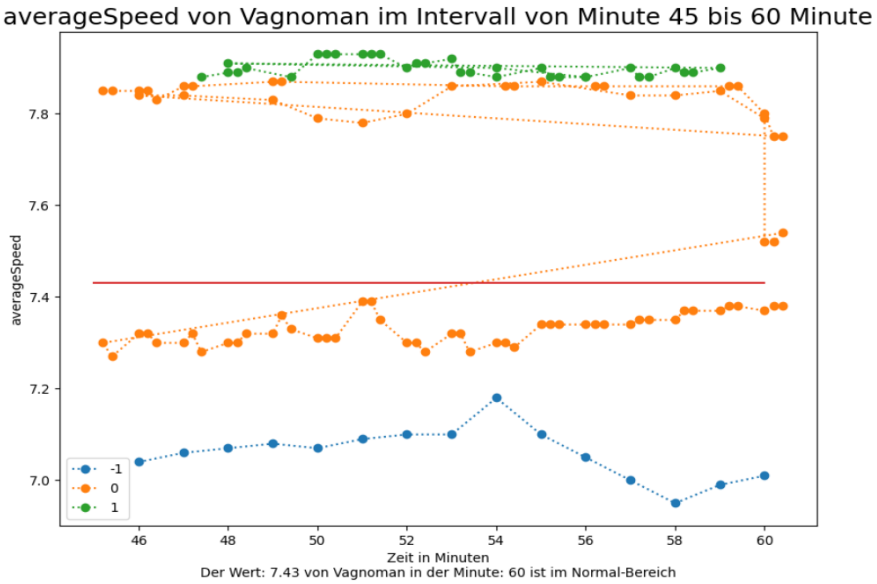

Abbildung 62: Graph Ausgabe des Buttons "Graph ausgeben" von *GUI_Stadartabweichung.py* (Eigendarstellung)

Die Überschrift des Graphen enthält das Attribut und den Spieler, die ausgewählt wurden und das zur ausgewählten Spielminute zugehörige 15 Minuten Intervall. Die Unterschrift des Graphen enthält den erfragten Wert *W* und informiert schriftlich darüber, ob der Wert *W* durchschnittlich, überdurchschnittlich oder unterdurchschnittlich ist.

GUI_Aufstellung.py

Die zweite GUI übernimmt viele Funktionalitäten der ersten GUI. Sie basiert auf den Daten der *dfldata*. Diese GUI zeigt dem Nutzer ein Fußballfeld auf, auf welchem vier Spielernamen in Button-Form angezeigt werden.

Abbildung 63: Screen der *GUI_Aufstellung.py* nach Betätigung des "Werte aktualisieren" Buttons (Eigendarstellung)

Diese sind initial als weiße Buttons dargestellt. Darunter findet der Nutzer einen Button und zwei Option-Menus. In dem ersten Options-Menu kann das Attribut gewählt werden, welches man abfragen will, in dem darunterliegenden, die aktuelle Spielminute. Der Button über den beiden Option-Menus mit der Inschrift „Werte aktualisieren“ fragt die Datenbank *dfldata* nach

dem Attribut, welches in der ersten Options-Menu ausgewählt wurde, für alle Spieler, die in den Buttons dargestellt sind, ab. Dieser abgefragte Wert wird daraufhin für jeden Spieler einzeln anhand der jeweiligen Standardabweichungen, welche aus dem, zur ausgewählten Minute gehörigen, Minuten-Intervall berechnet wurden, klassifiziert. Je nachdem zu welchem Ergebnis es hierbei kommt wird der Button der Spieler entweder Rot (unter der Standardabweichung), Grün (über der Standardabweichung) oder Gelb (innerhalb der Standardabweichung) eingefärbt. Dieser Vorgang kann beliebig oft wiederholt werden. Die Standardabweichung wird wie bereits in der vorherigen GUI basierend auf den Spieler spezifischen Dataframes *D* erstellt. Die Spielernamen sind momentan noch fest im Code definiert, auch die Position der Spieler muss noch im Code festgelegt und gesetzt werden. Daher muss dies für jeden Spieltag entsprechend angepasst werden. Die Spielminute muss auch mit angegeben werden, da diese nicht aus *dfldata* abgerufen werden kann.

5.1.2 Implementierung

Die GUIs nutzen das in 4.1.2 beschriebene Programm *MDB_Data*.py und das in 4.1.1 beschriebene Programm *connectMongo.py*.

GUI_Standardabweichung.py

Erstellt ein Fenster mit drei Options-Menus, drei Labels und zwei Buttons. Der Inhalt des ersten Options-Menus wird mit dem Rückgabewert von *connectMongo.get_defenders()* initialisiert. Das zweite Options-Menu wird mit einer Liste von Attributen als Strings initialisiert und das letzte mit einer Liste von Minuten als Zahlen. Die drei Labels enthalten jeweils eine Aufforderung zur Auswahl eines Wertes aus dem zugehörigen Options-Menu. Der Button „Beenden“ beendet das Programm. Der Button „Graph ausgeben“ ruft die Methode *button_action()* auf.

button_action(): Erfragt den gewählten Namen *name* des Spielers, die gewählte Minute *minute* und das gewählte Attribut *attribute* aus den aktuellen Inhalten der zugehörigen Options-Menus. Ruft die Methode *pick_df()* von *connectMongo.py* mit dem erfragten Namen auf und speichert sich den zum Spieler zugehörigen Dataframe als *df* und den weiteren Rückgabewert als *p_name*. Speichert sich die Rückgabe aus *connectMongo.connect()* unter der Variablen *players* ab. Ruft die Methode *get_value()* von *connectMongo.py* auf, übergibt dieser die Variablen

players, p_name und *attribute* und speichert die Rückgabe dieser als Variable *curr_val*. Ruft anschließend die Methode *give_graph()* von *connectMongo.py* mit den Variablen *df, attribute, minute, curr_val* und *p_name* auf.

GUI_Aufstellung.py

Erstellt ein Fenster mit fünf Buttons und zwei Options-Menus. Vier der Buttons sind in einem Fußballfeld platziert und beinhalten als Inschrift Spielernamen, sie werden statisch initialisiert. Der fünfte Button liegt unter dem Fußballfeld und trägt die Inschrift „Werte aktualisieren“. Das erste Options-Menu lässt den Nutzer eine Reihe an Attributen auswählen. Der Inhalt dieses Options-Menus wird mit einer Liste von Attributen als Strings initialisiert. Das zweite Options-Menu lässt den Nutzer eine Spielminute auswählen, es wird mit einer Liste von Zahlen von eins bis 90 initialisiert. Außerdem wird eine Liste *button_lst* initialisiert, die die vier Spieler-Buttons als Buttons beinhaltet. Wenn der Knopf „Werte aktualisieren“ gedrückt wird, wird die Methode *update_val()* aufgerufen, welche eine Reihe von Hilfsmethoden nutzt.

min_abfrage(player_name, att, but): Bekommt die Strings *player_name* und *att* übergeben sowie den Button *but*. Speichert in der Variablen *players* den Rückgabewert von *connectMongo.connect()*, in der Variablen *new_val* den Rückgabewert aus *connectMongo.get_value* und in der Variablen *col* den Rückgabewert aus *connectMongo.give_color_cluster()*. Setzt die Farbe des Buttons but auf die Farbe, welche in der Variablen *col* gespeichert wird.

update_helper(but, ds, attr): Bekommt einen Button *but*, eine Liste *ds* und einen String *att* übergeben. Ruft für den Spielernamen in der Liste *ds*, welcher in dem Button *but* als Text gespeichert ist, die Methode *min_abfrage()* auf und übergibt dieser zusätzlich *att* und *but*.

update_val(): Speichert unter der Variablen *ds* den Rückgabewert aus *connectMongo.get_defenders()*. Speichert unter *attr* das aktuell im zugehörigen Options-Menu stehendes Attribut als Sting ab. Ruft für jeden Button in der Liste *button_lst* die Methode *update_helper()* auf und übergibt an diese den jeweiligen Button *but* sowie *ds* und *attr*.

5.2 GUI Klassifikatoren

Diese beiden GUIs wurden, wie auch die vorherigen beiden, mit TKinter erstellt. Hierbei bezieht *GUI_MongoDB.py* ihre Daten aus *MDB_Data.py* und *GUI_DFL.py* bezieht ihre Daten aus *DFL_Data.py*, diese sind in 4.1.2 beschrieben. Die ebenfalls von den Programmen *GUI_MongoDB.py* und *GUI_DFL.py* genutzten Programme *classifier_MDB.py* und *classifier_DFL.py* sind ebenso in 4.1.2 beschrieben.

5.2.1 GUIs

GUI_MongoDB.py

Diese GUI funktioniert ähnlich wie die zweite GUI aus 5.1.1 und sieht auch ähnlich aus.

Abbildung 64: Screen der *GUI_MongoDB.py* nach Betätigung des "Werte aktualisieren" Buttons (Eigendarstellung)

Sie fragt bei druck des Buttons „Werte aktualisieren“ ebenfalls die Datenbank *dfldata* nach dem aktuellen Wert der, in den Buttons gelisteten, Spieler ab. Dieser Wert wird dann dem Programm *classifier_MDB.py* übergeben, welches anhand der im zweiten Options-Menu gewählten Minute das passende Minuten-Intervall mit KMeans und sieben Clustern clustert. Auf Basis dieser Clusterung wird dann ein Klassifikator gebaut und es wird eine Vorhersage zurückgegeben, in welchem Cluster und damit Bereich sich der abgefragte Wert im Kontext der ausgewählten Minute befindet. Je nachdem, ob der Wert als „Gut“, „Schlecht“ oder „Durchschnittlich“ klassifiziert wurde, wird er grün, rot oder gelb eingefärbt.

Diese GUI kann, während eines Live-Spiels, aufgerufen werden und wird dann die aktuellen Daten der Spieler abfangen, jedoch muss die aktuelle Spielminute immer angegeben werden, da diese nicht aus der Datenbank *dfldata* abgefragt werden kann.

GUI_DFL.py

Diese GUI zeigt, wie auch die vorherige, ein Fußballfeld mit vier Buttons in welchen Spielernamen stehen. Auch hier sind die Buttons initial weiß eingefärbt.

Abbildung 65: Screen der *GUI_DFL.py* nach Betätigung des "Werte aktualisieren" Buttons (Eigendarstellung)

Im Gegensatzt zur vorherigen GUI ist hier nur ein Options-Menu vorhanden, in welchem das Attribut ausgewählt werden kann, welches abgefragt werden soll. Über dem Options-Menu ist ein Label plaziert, welches die aktuelle Spielminute aufzeigt. Über dem Label ist der Button „Werte aktualisieren“ plaziert. Beim betätigen dieses wird der aktuelle Wert des ausgewählten Attributes, für jeden Spieler aus den Buttons, aus den DFL-URLs abgefragt. Es erfolgt dann für jeden Spieler die Clusterung der bisher aufgezeichneten Werte dieses Attributes in dem zur aktuellen Spielminute passenden Minuten-Intervall. Auf Basis dieser Clusterung wird dann ein Klassifikator gebaut, welcher eine Vorhersage darüber trifft, in welchem Cluster und damit Bereich sich der abgefragte Wert befindet. Je nachdem, ob sich der abgefragte Wert im Bereich

„Gut“, „Schlecht“ oder „Durchschnittlich“ befindet wird der Button grün, rot oder gelb eingefärbt.

Diese GUI kann ebenfalls, während eines Live-Spiels aufgerufen werden und wird dann die aktuellen Daten der Spieler abfangen, im Gegensatz zur vorherigen GUI muss dafür nicht die Spielminute explizit angegeben werden.

5.2.2 Implementierung

match_reader.py

Programm, welches das Auslesen der DFL-URLs übernimmt. Das Program *GUI_DFL.py* nutzt die Methoden *get_url(), get_root(), get_matchday(), defender_dicts(), get_clean_minute(), first_four_defenders()* und *get_player_val().* Benutzt außerdem indirekt die Hilfsmethoden *filter_team(), get_player_dicts()*

get_url(var): Bekommt einen String übergeben. Gibt abhängig von *var* entweder den URL der Positionsdaten oder den URL der Ereignisdaten zurück.

get_root(url): Bekommt eine URL als Sting übergeben und parst diese URL als Element-Tree. Gibt den Root dieses Element-Trees zurück.

get_matchday(root): Bekommt eine Root eines Element-Trees übergeben und gibt den Wert des Trees zurück, welcher unter dem Tag *MatchDay* steht.

filter_team(root, team_name): Bekommt eine Root *root* eines Element-Trees und einen String *team*_name übergeben. Gibt den Teil des unter *root* stehenden Trees zurück, welcher den String *team_name* im Tag enthält.

get_player_dicts(team_tree): Bekommt einen Teil *team_tree* eines Element-Trees übergeben. Speichert eine Reihe an Attributen und ihre zugehörigen Werten aller unter *team*_tree liegenden Tags in je ein Dictionary. Speicher alle diese Dictionarys in einer Liste und gibt die Liste zurück.

defender_dicts(): Ruft die Methode *filter_team()* auf und speichert den Rückgabewert in der Variablen *hsv*. Ruft die Methode *get_player_dicts()* auf und speichert den Rückgabewert in der

Variablen *player_dicts*. Filtert aus allen Elementen in *player_dicts* diejenigen heraus, die auch in der vordefinierten Liste *defs* enthalten sind und gibt die bearbeitete Variable *player_dicts* zurück.

get_clean_minute(root): Bekommt eine Root *root* eines Element-Trees übergeben. Speichert den Wert des Trees, welcher unter dem Tag *MinuteOfPlay* steht in der Variablen *minute*. Bereinigt diese Variable von Punkten und Pluszeichen und gibt diese bearbeitete Version von *minute* zurück.

first_four_defenders(): Ruft die Methode *defender_dicts()* auf und speichert den Rückgabewert in der Variablen *dic*. Speichert die ersten vier Einträge von *dic* in der Variablen *reduced_dic* und gibt diese zurück.

get_player_val(player-name, att_name): Bekommt die Strings *player_name* und *att_name* übergeben. Ruft die Methode *defender_dicts()* auf und speichert den Rückgabewert in der Variablen *def_dicts*. Gibt für den in *player_name* enthaltenen Spielernamen und das in *att_name* enthaltene Attribut den zugehörigen Wert in *def_dicts* zurück.

GUI_MongoDB.py

Erstellt ein Fenster mit fünf Buttons und zwei Options-Menus. Vier der Buttons sind in einem Fußballfeld platziert und beinhalten als Inschrift Spielernamen, sie werden statisch initialisiert. Der fünfte Button liegt unter dem Fußballfeld und trägt die Inschrift „Werte aktualisieren“. Das erste Options-Menu lässt den Nutzer eine Reihe an Attributen auswählen. Der Inhalt dieses Options-Menus wird mit einer Liste von Attributen als Strings initialisiert. Das zweite Options-Menu lässt den Nutzer eine Spielminute auswählen, es wird mit einer Liste von Zahlen von eins bis 90 initialisiert. Außerdem wird eine Liste *button_lst* initialisiert, die die vier Spieler-Buttons als Buttons beinhaltet. Wenn der Knopf „Werte aktualisieren“ gedrückt wird, wird die Methode *update_val()* aufgerufen, welche eine Reihe von Hilfsmethoden nutzt.

give_color(klasse): Bekommt eine Zahl *klasse* übergeben. Wenn *klasse* gleich eins ist wird der Hexadezimalcode für die Farbe Grün zurückgegeben, wenn *klasse* gleich minus eins ist wird der Hexadezimalcode für die Farbe Rot zurückgegeben und wenn *klasse* gleich null ist wird

der Hexadezimalcode für die Farbe Gelb zurückgegeben. Alle Hexadezimalcodes werden als String zurückgegeben.

min_abfrage(player_name, att, but): Bekommt die Strings *player_name* und *att* übergeben sowie den Button *but*. Speichert in der Variablen *players* den Rückgabewert von *connectMongo.connect()*. In der Variablen *new_val* wird der Rückgabewert aus *connectMongo.get_value()* gespeichert, dabei wird der Methode *players*, *att* und *player_name* übergeben. In der Variable *klasse* wird der Rückgabewert aus *classifier_MDB.give_class()* gespeichert, dabei wird der Methode *player_name*, *att*, eine Liste aus dem Wert des Minuten-Option-Menus und *new_val*, eine boolesche Variable und ein String *classifier* übergeben. In der Variablen *col* wird der Rückgabewert aus *give_color()* gespeichert, wobei der Methode die Variable *klasse* übergeben wird. Setzt die Farbe des Buttons but auf die Farbe, welche in der Variablen *col* gespeichert wird.

update_helper(but, ds, attr): Bekommt einen Button *but*, eine Liste *ds* und einen String *att* übergeben. Ruft für den Spielernamen in der Liste *ds*, welcher in dem Button *but* als Text gespeichert ist die Methode *min_abfrage()* auf und übergibt dieser zusätzlich *att* und *but*.

update_val(): Speichert unter der Variablen *ds* den Rückgabewert aus *connectMongo.get_defenders()*. Speichert unter *attr* das aktuell im zugehörigen Options-Menu stehendes Attribut als Sting ab. Ruft für jeden Button in der Liste *button_lst* die Methode *update_helper()* auf und übergibt an diese den jeweiligen Button *but* sowie *ds* und *attr*.

GUI_DFL.py

Erstellt ein Fenster mit fünf Buttons, einem Label und einem Options-Menu. Vier der Buttons sind in einem Fußballfeld platziert und beinhalten als Inschrift Spielernamen, sie werden durch *match_reader.def_dicts()* initialisiert. Der fünfte Button liegt unter dem Fußballfeld und trägt die Inschrift „Werte aktualisieren“. Der Inhalt des Labels wird mit *match_reader.get_clean_minute()* initialisiert. Das Options-Menu lässt den Nutzer eine Reihe an Attributen auswählen. Der Inhalt dieses Options-Menus wird mit einer Liste von Attributen als Strings initialisiert. Außerdem wird die Variable *url* mit *match_reader.get_url()* initialisiert, die Variable *root* mit *match_reader.get_root()* und die Variable *player_dicts_lst* mit *match_reader.first_four_defs()*. Zusätzlich wird eine Liste *button_list*, die die vier Spieler-Buttons als

Buttons beinhaltet initialisiert. Wenn der Knopf „Werte aktualisieren“ gedrückt wird, wird die Methode *update_val()* aufgerufen.

update_val(): Speichert unter der Variablen *root* den Rückgabewert von *match_reader.get_root(match_reader.get_url())* und unter der Variablen *minute* den Rückgabewert aus *match_reader.get_clean_minute()*. Für jeden Button in der *button_list* und für jedes Dictionary aus *players_dicts_list* wird die Methode *match_reader.get_player_val()* mit dem Spielernamen und dem aktuellen Inhalt des Options-Menus ausgeführt und in der Variable *att_val* gespeichert. Daraufhin wird in der Variablen *klasse* der Rückgabewert aus der Methode *classifier_DFL.classify_with_threshold()* gespeichert. Die übergebenen Parameter bestehen aus dem Inhalt des Options-Menus, einer Liste aus *minute* und *att_val*, zwei Stings, die den Klassifikator und das Clustering-Verfahren bestimmen und einer booleschen Variable. Anhand des Wertes in *klasse* wird daraufhin für den jeweiligen Button die Farbe des Buttons angepasst. Zusätzlich wird am Ende noch der Inhalt des Labels angepasst, indem der Inhalt dem Rückgabewert der Methode *match_reader.get_clean_minute()* gleichgesetzt wird.

6 Fazit

Es stellte sich heraus, dass sich die Rohdaten aus den DFL-URLs besser für diese Zwecke der Klassifizierung eignen, da es sich eben um Rohdaten handelt und diese nicht noch Berechnungen unterzogen werden, wie es bei den Daten von *dfldata* der Fall ist. Außerdem enthalten die DFL-URL Rohdaten eine höhere Anzahl an Attributen, die abgefragt werden können. So ermöglicht dies, Attribute in Betracht zu ziehen, welche, abhängig von der Position der Spieler, interessanter und sinnvoller für die jeweiligen Spieler sind. Da es in dieser Arbeit um die Verteidiger des Teams ging, waren Attribute wie *shotsAtGoal* oder *assistsShotsAtGoal* aus der *dfldata* weniger interessant. Diese brachten beim Clustering und der nachfolgenden Klassifikation auch weniger sinnvolle Ergebnisse, da sie fast durchgehend einen Wert von null aufwiesen. Bei den DFL-URLs können jedoch Attribute wie *tackelingsAirWon* oder auch

defenciveClearances abgefragt werden, welche in diesem Fall einen sinnvolleren Maßstab für die Leistung der Verteidiger bilden würden. Dementsprechend könnte durch die Nutzung verschiedener Attribute für verschiedenen Positionen eine bessere Beurteilung der Leistungen aller Spieler geschehen.

Allerdings wurden in dieser Arbeit für das Clustering und die Klassifikation der DFL-URL Rohdaten die Verteidiger verschiedenen Teams aufgezeichnet, da ansonsten zu wenig Daten zur Verfügung stehen würden. Daher wurden die Cluster für jeden Spieler auf demselben Datensatz gebildet, dementsprechend werden für jeden Spieler dieselben Bereiche „Gut", „Schlecht" und „Durchschnittlich" erstellt. Dies ist nicht optimal, da auch die Verteidiger unter sich verschiedenen Positionen und somit auch verschiedene Aufgaben übernehmen und die persönlichen Leistungen von Spieler zu Spieler variieren. Es sollte bei den Rohdaten der DFL-URL, ähnlich wie bei den *dfldata* Daten, die Clusterung und Klassifikation für jeden Spieler spezifisch ablaufen. Damit dies gewährleistet werden kann müssen mehr Daten gesammelt werden, also Aufzeichnungen der Spieler über mehrere Spieltage erfolgen. Dadurch kann pro Spieler ein Dataframe entstehen, welcher dann für den zugehörigen Spieler die Basis der Clusterung und der Klassifikation bilden kann. Somit würde jeder Spieler auf Basis seiner eigenen vorhergegangenen Leistungen beurteilt werden.

Aufgrund der, beim Clustering entdeckten, Ausreißer wäre eine Ausreißer Behandlung in Zukunft wohl sinnvoll. Dies könnte zu noch besseren Ergebnissen beim Clustering führen und damit auch bessere Ergebnisse in der Klassifikation liefern. Allerdings sollte auch hier erst auf größere Datensätze gewartet werden, um nicht vorschnell Werte als Ausreißer anzusehen.

In Zukunft sollte auch auf MeanShift als Clustering-Verfahren verzichtet werden, denn dieses Verfahren liefert schlechtere Ergebnisse in der Klassifikation und dauert, aufgrund der hohen Anzahl der gebildeten Cluster, länger als KMeans. Die Ergebnisse der Genauigkeit der Klassifikatoren weisen, mit dem bisherigen Datensatz, alle ein gewisses Overfitting auf, hier benötigt es daher ebenfalls eines noch größeren Datensatzes. Wenn die Clusterung in Zukunft spielerspezifisch ablaufen soll, werden hier die Ergebnisse aufgrund der kleineren Datensätze noch länger vom Overfitting betroffen sein. Mit größeren Datensätzen können in Zukunft auch Neuronale Netze zur Klassifikation genutzt werden, um bessere Ergebnisse zu ermöglichen. Aufbauend auf diesen besseren Ergebnissen könnten dann Bewertungen der Spieler und ihrer

Leistungen an den jeweiligen Spieltagen erstellt werden, unteranderem könnte dadurch ein besseres Verständnis der aktuellen Form der Spieler entstehen.

Literaturverzeichnis

Armatas, V., Yiannakos, A., & Sileloglou, P. (2007). Relationship between time and goal scoring in soccer games: Analysis of three World Cups. *International Journal of Performance Analysis in Sport, 7*(2), S. 48-58. doi:10.1080/24748668.2007.11868396

Bialkowski, A., Lucey, P., Carr, P., Yue, Y., Sridharan, S., & Matthews, I. (2014). Large-Scale Analysis of Soccer Matches Using Spatiotemporal Tracking Data. *2014 IEEE International Conference on Data Mining*, S. 725-730. doi:10.1109/ICDM.2014.133

Boluki, S., Zamani Dadaneh, S., & Qian, X. (2019). Optimal clustering with missing values. *BMC Bioinformatics 20*. Von https://doi.org/10.1186/s12859-019-2832-3 abgerufen

Bradley, P., & Noakes , T. (2013). Match running performance fluctuations in elite soccer: Indicative of fatigue, pacing or situational influences? *Journal of Sports Sciences, 31*(15), S. 1627-1638. doi:10.1080/02640414.2013.796062

Brefeld, U., Davis, J., Van Haaren, J., & Zimmermann, A. (2019). *Machine Learning and Data Mining for Sports Analytics* (1 Ausg.). Springer International Publishing.

Brooks, J., Kerr, M., & Guttag, J. (2016). Using machine learning to draw infer-ences from pass location data in soccer. *Statistical Analysis and Data Mining: The ASA Data Science Journal, 9*. Von https://doi.org/10.1002/sam.11318 abgerufen

Brownlee , J. (2020). *Machine Learning Mastery*. Abgerufen am 10. Februar 2021 von https://machinelearningmastery.com/clustering-algorithms-with-python/

Chen , S.-C., Shyu, M.-L., Chen, M., & Zhang , C. (2004). A decision tree-based multimodal data mining framework for soccer goal detection. *2004 IEEE International Conference on Multimedia and Expo (ICME) (IEEE Cat. No.04TH8763), 1*, S. 265-268. doi:10.1109/ICME.2004.1394176

Data to Fish. (2020). Abgerufen am 15. Januar 2021 von https://datatofish.com/k-means-clustering-python/

DFL Deutsche Fußball Liga GmbH. (2018/2019). Definitionskatalog Offizielle Spieladaten. *5.0*.

Di Salvo, V., Baron, R., Gonzlez-Haro, C., Gormasz, C., Pigozzi, F., & Bachl, N. (2010). Sprinting analysis of elite soccer players during European Champions League and UEFA Cup matches. *Journal of Sports Sciences, 28*(14), S. 1489-1494. Von http://dx.doi.org/10.1080/02640414.2010.521166 abgerufen

Girgin, S. (2019). *Medium*. Abgerufen am 19. Januar 2021 von https://medium.com/@sametgirgin/hierarchical-clustering-model-in-5-steps-with-python-6c45087d4318

Jones, P., James, N., & Mellalieu, S. (2004). Possession as a performance indicator in soccer. *International Journal of Performance Analysis in Sport, 4*(1), S. 98-102. doi:10.1080/24748668.2004.11868295

Lago-Peñas, C., Rey, E., Lago-Ballesteros, J., Casais, L., & Domínguez, E. (2009). Analysis of work-rate in soccer according to playing positions. *International Journal of Performance Analysis in Sport, 9*(2), S. 218-227. Von https://doi.org/10.1080/24748668.2009.11868478 abgerufen

Link , D. (2018). Data Analytics in Professional Soccer: Performance Analysis Based on Spatiotemporal Tracking Data. Springer Vieweg.

Link, D., & de Lorenzo, M. (2016). Seasonal Pacing - Match Importance Affects Activity in Professional Soccer. *PLOS ONE, 11*(6). Von https://doi.org/10.1371/journal.pone.0157127 abgerufen

Madhulatha, T. (2012). An Overview on Clustering Methods. *IOSR Journal of Engineering, 2*, S. 719-725. doi:10.9790/3021-0204719725

Memmert, D., Lemmink, K., & Sampaio, J. (2017). Current Approaches to Tactical Performance Analyses in Soccer Using Position Data. *Sports medicine, 47*(1), S. 1–10. Von https://doi.org/10.1007/s40279-016-0562-5 abgerufen

Patel, K., & Thakral, P. (2016). The best clustering algorithms in data mining. *2016 International Conference on Communication and Signal Processing (ICCSP)*, S. 2042-2046. doi:10.1109/ICCSP.2016.7754534

Perin, C., Vuillemot, R., Stolper, C., Stasko, J., Wood, J., & Carpendale, S. (2018). State of the Art of Sports Data Visualization. *Computer Graphics Forum, 37,* S. 663-686. Von https://doi.org/10.1111/cgf.13447 abgerufen

Płoński, P. (2020). *MLJAR Automated Machine Learning*. Abgerufen am 23. April 2021 von https://mljar.com/blog/visualize-decision-tree/

python. (kein Datum). Abgerufen am 14. Februar 2021 von https://docs.python.org/3/library/tkinter.html

Rein, R., & Memmert, D. (2016). Big data and tactical analysis in elite soccer: future challenges and opportunities for sports science. *SpringerPlus, 5*(1). Von https://doi.org/10.1186/s40064-016-3108-2 abgerufen

scikit learn. (kein Datum). Abgerufen am 15. Februar 2021 von https://scikit-learn.org/stable/modules/clustering.html

scikit learn. (kein Datum). Abgerufen am 18. März 2021 von https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html

scikit learn. (kein Datum). Abgerufen am 19. März 2021 von https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier

scikit learn. (kein Datum). Abgerufen am 21. März 2021 von https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

Seif El-Nasr, M., Drachen, A., & Canossa, A. (2013). *Game Analytics: Max-imizing the Value of Player Data* (1 Ausg.). Springer-Verlag London.

Van Haaren, J., Dzyuba, V., Hannosset, S., & Davis, J. (2015). Automatically Discovering Offensive Patterns in Soccer Match Data. In *Advances in Intelligent Data Analysis XIV* (S. 286-296). Springer, Cham. Von https://doi.org/10.1007/978-3-319-24465-5_25 abgerufen

Venkatkumar, I., & Shardaben, S. (2016). Comparative study of data mining clustering algorithms. *2016 International Conference on Data Science and Engineering (ICDSE)*, S. 1-7. doi:10.1109/ICDSE.2016.7823946

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original