FH | JOANNEUM
University of Applied Sciences

Sparking the future

Autoren
Rauchenwald Felix, Schneider Lukas

Betreuer
FH-Prof. Dipl.-Ing. Dr.techn. Peter Salhofer

FH – JOANNEUM
Informationsmanagment

27.01.18

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

Quellen:
https://www.tutorialspoint.com/r/r_linear_regression.htm,
https://www.slideshare.net/ittalk/apache-spark-overview-69310245,
https://de.wikipedia.org/wiki/K-Means-Algorithmus

CODEVERZEICHNIS

1 EINLEITUNG

Dieses Dokument beinhaltet die Dokumentation zu unserer bereichsübergreifenden Projektarbeit „Sparking the Future“ für das 5. Semester des Studienganges Informationsmanagement, auf der FH Joanneum.
Das Ziel dieser Arbeit war es, die Möglichkeiten des Apache Spark Frameworks zu untersuchen und die Erkenntnisse zu dokumentieren. Dabei wurde explizit keine konkrete Fragestellung festgelegt, stattdessen sollte es darum gehen, dass wir uns einen Überblick verschaffen und uns das zugehörige Wissen aneignen.
Es wurden Anhaltspunkte festgelegt, welche eine ungefähre Orientierung im sehr umfangreichen Themenbereich Machine Learning geben sollten. So sollten die Medientransparenzdaten auf politische Auffälligkeiten untersucht werden.
Weiters sollten unterschiedliche Algorithmen zu Regression, Klassifikation, Clustering und Decision Trees des Spark Framework näher betrachtet werden.
Diese enorme Möglichkeitsvielfalt erwies sich als Vor- und Nachteil zugleich, denn Aufgrund der enormen Spannweite des Themenbereiches Machine Learning, musste zwangsläufig auf viele Themenbereiche verzichtet werden. Diese Selektierung erlaubte es uns allerdings, einen Überblick über dieses massive und spannende Thema und dessen Teilbereiche zu erhalten.
Ohne viel vorwegnehmen zu wollen, sind wir im Laufe unserer Arbeit auf einige interessante Erkenntnisse in puncto Verspätungsursachen von Linienflügen, Finanzierungsformen von Ministerien und der Punktezahl von Basketballspielern gestoßen.
Damit nicht nur wir, sondern auch alle möglichen Interessenten an unseren Erkenntnissen teilhaben können, haben wir alle öffentlichen Datensätze, unseren geschriebenen Code, sowie die Resultate unserer Arbeit auf Github unter

https://github.com/MrOrange1992/SparkML

öffentlich zur Verfügung gestellt.
Wir wünschen allen Interessierten viel Freude beim Lesen unserer Dokumentation - möge der Funke an Begeisterung zu diesen spannenden und umfangreichen Themen, welcher uns beim Verfassen dieser Arbeit erwischt hat, auch an den Leser/die Leserin überspringen.

Felix Rauchenwald & Lukas Schneider

1.1 GRUNDBEGRIFFE

Zunächst werden einige Grundbegriffe definiert, welche zum Verstehen dieser Arbeit unbedingt notwendig sind.

1.1.1 Data Science

Unter Data Science versteht man die Verknüpfung mehrerer, klassischer Disziplinen der Mathematik, wie Statistik und Algorithmik - mit Techniken von Software Engineering, wie Programmierung und Machine Learning.
Das Ziel ist es, große Mengen an Daten zu untersuchen und Erkenntnisse daraus zu gewinnen.

1.1.2 Machine Learning

Beim Machine Learning werden Datensätze analysiert und ausgewertet, um sich wiederholende Muster zu identifizieren, und diese sinnvoll zu verarbeiten. Dies geschieht weitgehend autonom, durch einen, mit passender Software ausgestatteten, Computer. Diese Definition ist absichtlich sehr weit formuliert, da viele unterschiedliche Algorithmen und Herangehensweisen im Bereich Machine Learning existieren, welche aber unterschiedliche Ziele verfolgen. Somit ist Machine Learning eher als Schirmbegriff, als ein engbestimmter Themenbereich zu sehen.
Die Zwecke des Machine Learning umfassen Informationsextraktion, Prognosen, Optimierung, sowie die autonome Entscheidungsfindung.

1.1.3 Supervised Learning

Unter "Supervised Learning" (deutsch: "Überwachtes Lernen") versteht man eine Methode des Machine Learning, durch die der Algorithmus unter "Aufsicht" trainiert wird bzw lernt. Dabei wird vom Benutzer festgelegt für welche Input Daten vom Algorithmus eine Funktion angelernt werden soll, welche nach einem festgelegten Schema die Ouput Daten vorhersagt. Dabei wird zwischen Klassifikation und Regression unterschieden. Ein Klassifikationsproblem beschreibt die Notwendigkeit die Ergebnisse in eine spezifizierte Anzahl an Klassen aufzulösen. Ist der Bereich für Ergebnisse jedoch fortlaufend spricht man von einem Regressionsproblem. Apache Spark stellt eine Vielzahl an Algorithmen für beide Problemstellungen zu Verfügung.

1.1.4 Unsupervised Learning

"Unsupervised Learning" (deutsch: "Unüberwachtes Lernen") unterscheidet sich von Überwachtem Lernen dadurch, dass Daten keine Label mit richtigen oder gewünschten Ergebnissen bekommen, sondern nach bis dato unbeobachteten Mustern oder Regelmäßigkeiten gesucht wird. Da durch das Fehlen eines Labels auch kein Ziel gesetzt wird, kann auch keine Genauigkeit des Outputs bestimmt werden.
Techniken des Unsupervised Learning umfassen Clustering, Anomaly Detection und Neuronale Netzwerke.

1.2 Apache Spark

1.2.1 Einleitung

Apache Spark ist ein open-source Framework für zusammengeschlossene Computersysteme, sogenannte Cluster. Es besteht jedoch die Möglichkeit, es auf einzelnen Rechnern einzusetzen. Es stellt APIs für die Programmiersprachen Java, Scala, Python und R zur Verfügung, läuft auf Windows, OS X und Linux Betriebssystemen und wird aktuell von der Apache Software Foundation entwickelt. Die aktuelle Version ist 2.3.0 (Stand 12.03.18).
In unserer Arbeit wurde die Version 2.2.0 von Apache Spark aus Kompatibilitätsgründen mit Java in der Version 8 und Scala 2.10.6 eingesetzt.
Spark setzt sich aus mehreren, miteinander verknüpften, Komponenten zusammen.

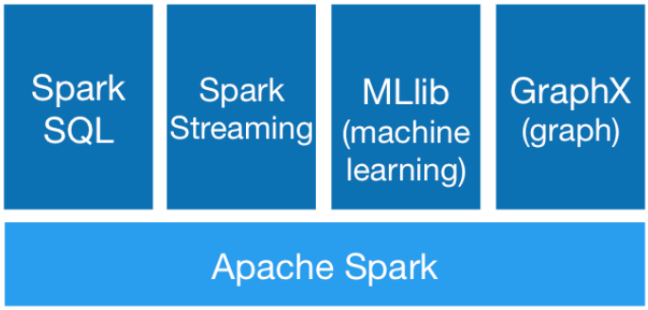

Abbildung 1 - Apache Spark Komponenten

Die Basis (in Hellblau abgebildet) stellt der Spark Core dar, welche die grundlegenden Funktionalitäten wie Aufgabenverteilung, Scheduling usw. zur Verfügung stellt (vgl. [https://de.wikipedia.org][1]). Zur Datenhaltung werden sogenannte RDDs oder Dataframes eingesetzt.
Die Spark SQL Komponente erlaubt es, mit DataFrames zu arbeiten. Darüber hinaus stellt Spark SQL-sprachenspezifische Befehle zur Verfügung, um mit DataFrames nativ in Scala, Java und Python umgehen zu können. DataFrames erlauben im Vergleich zu RDDs, ein benutzerfreundlicheres und Dank der vorhandenen SQL-Ähnlichkeit, vertrauteres Manipulieren von Daten.
Spark Streaming ermöglicht eine nahe Echtzeitanalyse für zeitkritische Auswertungen, wie Sensordaten. Auf Spark Streaming wird in dieser Arbeit nicht näher eingegangen.
MLlib Machine Learning Library ist ein Framework, welches auf den Spark Core aufsetzt. Es ist darauf ausgelegt, alle relevanten Daten im Arbeitsspeicher zu halten, wodurch es performanter als die meisten vergleichbaren Frameworks ist. Viele Machine Learning und Statistikalgorithmen sind implementiert, wie die in dieser Arbeit verwendeten Algorithmen für Logistische und Lineare Regression, Entscheidungsbäume und K-means.
Die GraphX Komponente ist ein Framework zur Verarbeitung von Graphen. Wie Spark Streaming, wird GraphX an dieser Stelle nur der Vollständigkeit wegen erwähnt, in der weiteren Arbeit wird nicht näher darauf eingegangen.

1.2.2 RDDs

RDD – kurz für „Resiliant distributed dataset“ ist ein Datentyp bzw. eine Klasse, mit denen Daten in den SparkContext eingelesen bzw. weiterverarbeitet werden können. Um eine simple Collection an Daten in den SparkContext einzulesen, stellt dieser eine eigene Funktion zur Verfügung, um diese zu parallelisieren. Der Gedanke hinter RDDs ist das Partitionieren von Daten, um diese weiterführend dynamisch und parallel über ein Cluster und dessen Worker zu

verteilen, was den Grundbaustein des Spark Frameworks darstellt. Um dies umsetzen zu können, ist die Voraussetzung für RDDs deren Unveränderbarkeit als Eigenschaft des Datentyps.

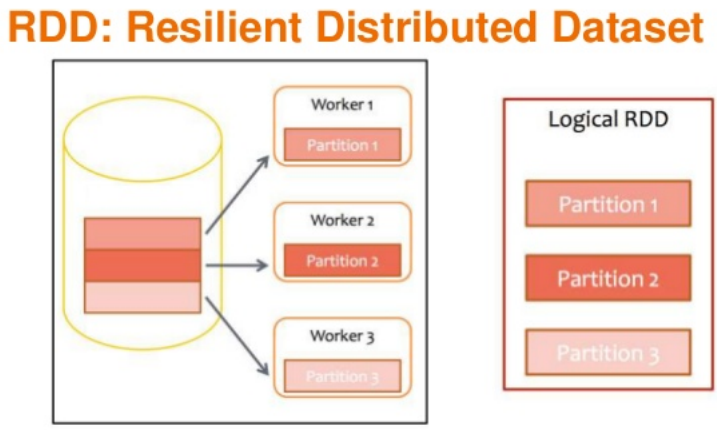

Abbildung 2 - RDD Schema

Dank dieser Unveränderbarkeit bieten RDDs alle gängigen Operationen der funktionalen Programmierung die von Scala, der nativen Sprache von Spark unterstützt werden.

1.2.3 DataFrames

Dataframes stellen eine auf RDDs aufgesetzte Abstraktion dar. Diese sind aufgrund ihrer Struktur mit der Tabelle einer relationalen Datenbank vergleichbar. Weiters haben die Operationen eines DataFrames eine sehr SQL ähnliche Syntax. Sie eignen sich hervorragend Daten, die bereits in strukturierter Form vorhanden sind, in den SparkContext einzulesen. Die einzelnen Spalten eines DataFrames können bequem über deren Namen angesprochen und manipuliert werden. Neben allen gängigen SQL Statements wie *SELECT*, *WHERE*, *DROP*, *JOIN* usw., die als Operationen eingesetzt werden können, stellen sie auch funktionale Grundfunktionen zur Verfügung, wie zum Beispiel *FILTER*, die in Bezug auf *WHERE* jedoch redundant ist (vgl. [Ryza, Laserson, Owen, Wills 2017], S. 23).

1.2.4 Libraries

In der aktuellen Version von Spark gibt es zwei Implementierungen von Machine Learning Libraries. Diese sind das ältere *spark.mllib* Paket und das neuere *spark.ml* Paket. Mllib beinhaltet die originale API, welche auf RDDs aufbaut, ML stellt eine API zu Verfügung welche mit DataFrames arbeitet und das Arbeiten mit Pipelines ermöglicht. Pipelines ermöglichen mehrere Datentransformationen hintereinander anzuwenden, ohne dass diese erst verspeichert werden müssen.
Es wird die Verwendung von ML empfohlen, da DataFrames flexibler und vielseitiger als RDDs sind, jedoch wird auch die Mllib weiterhin unterstützt, welche Dank der längeren Entwicklungszeit mehr Features bietet. Langfristig ist geplant, Mllib vollständig durch ML zu ersetzen (vgl. [https://spark.apache.org][1]).
Für unser Projekt wurden beide Libraries verwendet und sofern angebracht, miteinander verglichen (siehe Decision Tree – NBA Datensatz)

1.3 DATENSÄTZE UND DATENHALTUNG

1.3.1 Medientransparenz

Es wurden Datensätze bestehend aus Zahlungen von Organisationen von diversen Medien in Österreich von https://www.medien-transparenz.at bezogen, um politische Zusammenhänge anhand dieser Zahlungen zu analysieren. Die Einträge des Datensatzes beziehen sich auf die Zeitspanne von Q3 2012 bis Q2 2017. Der Spark SQLContext stellt eine *read* Methode zum Einlesen von CSV Daten zur Verfügung.

```
val dataFrame: sql.DataFrame = sparkSession.sqlContext.read
  .format("com.databricks.spark.csv")
  .option("header", "true")
  .schema(customSchema)
  .load("./dataFiles/MT-20123-20172.csv")
  .cache()
```

Code 1 - Einlesen der CSV Daten

Durch das Anlegen eines *customSchema* werden die Header der einzelnen Felder eingelesen und vom Text in den richtigen Datentypen gemappt.

```
val customSchema = StructType(Array(
  StructField("organisation", StringType, true),
  StructField("federalState", StringType, true),
  StructField("media", StringType, true),
  StructField("transferType", IntegerType, true),
  StructField("period", IntegerType, true),
  StructField("amount", FloatType, true)
))
```

Code 2 - Custom Schema zum Einlesen der Daten

Das Feld transferType beinhaltet den Typ der Zahlungen, die sich auf

- Zahlungen gemäß §2 MedKF-TG (Medien-Kooperationen)
- Zahlungen gemäß §4 MedKF-TG (Förderungen)
- Zahlungen gemäß §31 ORF-G (Gebühren)

beziehen. Dafür wurde eine Filter Funktion erstellt, um das DataFrame im Vorhinein nach Art der Zahlungen zu filtern.

```
def filterDF(mkp: Boolean, fdrg: Boolean, gbrn: Boolean): sql.DataFrame =
{
  if (mkp && fdrg && gbrn)
    dataFrame
  else if (mkp && fdrg && !gbrn)
    dataFrame.filter(dataFrame("transferType") =!= "31")
  else if (mkp && !fdrg && !gbrn)
    dataFrame.filter(dataFrame("transferType") =!= "31").filter(data-
Frame("transferType") =!= "4")
  else if (!mkp && fdrg && !gbrn)
    dataFrame.filter(dataFrame("transferType") =!= "31").filter(data-
Frame("transferType") =!= "2")
  else
    dataFrame
}
```

Code 3 - Filtern nach Art der Zahlung

1.3.2 NBA Statistik Daten

Für die Analyse wurden Decision Tree- und Linear-Regression-Algorithmen NBA Statistik Daten verwendet. Aufgrund der laufenden Saison (2017/18) und der sich dadurch täglich ändernden Daten, wurden diese dynamisch über API-Calls der MySportsFeeds API https://api.mysportsfeeds.com bezogen. Dafür wurde ein Account für die API erstellt und die Datensätze über HTTP-Requests mittels einer Mapper-Funktion bezogen.

Um dies zu bewerkstelligen, wurde eine Wrapper Klasse erstellt, um die im .csv Format kommenden Daten der API möglichst sauber in den SparkContext einzubinden. Da das gesamte Projekt auf Github gehostet ist, wurde in der IDE ein *sportsAPI.properties* File mit username und password für die API calls angelegt und dem *.gitignore* hinzugefügt.

```
def getProperty(property: String): Option[String] =
{
  Source.fromInputStream(getClass.getResourceAsStream("/sportsAPI.properties"))
    .getLines.find(_.startsWith(property)).map(_.replace(property + "=", ""))
}
```

Code 4 - SportsAPIWrapper GetProperty Function

```
//username and password for API requests
val apiKey: String = getProperty("username").get + ":" +  getProperty("pass-
word").get

//base URL path for API pull request
val pullURL: String = "https://api.mysportsfeeds.com/v1.1/pull/nba/"

//request only needed stats from API
val playerStatsRequest: String = "?play-
erstats=MIN/G,PTS/G,AST/G,REB/G,FT%25,FG%25"
```

Code 5 - Class Memeber APIWrapper

Mit einem http GET Request mittels Basic Authentication wird auf die API zugegriffen und die im Request enthaltenen Daten werden als InputStream eingelesen und als *List[String]* zwischengespeichert.

```
def getPlayerStatsOfSeason(season: String): List[String] = {
  try {
    val url = new URL(pullURL + season + "/cumulative_player_stats.csv" + play-
erStatsRequest)

    val encoding = Base64.getEncoder.encodeToString(apiKey.getBytes)
    val connection = url.openConnection.asInstanceOf[HttpURLConnection]
    connection.setRequestMethod("GET")
    connection.setDoOutput(true)
    connection.setRequestProperty("Authorization", "Basic " + encoding)
    val content = connection.getInputStream

    import java.io.{BufferedReader, InputStreamReader}
    val reader = new BufferedReader(new InputStreamReader(content))

    Stream.continually(reader.readLine()).takeWhile(_ != null).toList
  }
  catch { case e: Exception => e.printStackTrace(); List()}
}
```

Code 6 – Funktion für http Request

Die dadurch entstehende List[String] wird in den einzelnen Scala-Files in eine RDD[String] parallelisiert und dann weiter in ein DataSet gemappt. Die einzelnen Felder in der List[String] entsprechen den Parametern des HTTP-Request.

Tabelle 1 - NBA Daten Felder

Field name	Type	Comment
ID	Integer	Unique player ID
lastName	String	Player last name
firstName	String	Player first name
team	String	NBA team player is currently signed
position	Integer	Field position, mapped from String to Integer
height	Float	Player height
weight	Integer	Player weight
pointsPG	Float	Points per game
assistsPG	Float	Assists per game
reboundsPG	Float	Rebounds per game
fgPct	Float	Field goal percentage
ftPct	Float	Free throw percentage
minSecPG	Float	Active playing time per game
gamesPlayed	Integer	Total number of games played

Das DataSet wird mittels einer Case Class für Player aus der List[String] für die einzelnen Felder generiert.

Alle relevanten API calls bzw. mappings, auf die in dieser Dokumentation verwiesen wird, sind unter https://github.com/MrOrange1992/SparkML/blob/master/src/main/scala/DecisionTrees/WrapperMySportsAPI.scala zu finden.

1.3.3 Flugverkehr-Statistik-Daten

Die Daten für die Flugverkehr-Statistiken wurden vom Bureau of Transportation Statistics https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time im csv Format bezogen. Um den Datenumfang in handhabbarer Größe zu halten, beschränken sich die Einträge ausschließlich auf Kalifornien, USA in der Zeitspanne von Jänner 2017 bis März 2017. Um die gewünschten Felder aus dem Datensatz zu extrahieren, wurde ein Custom Schema erstellt, welches das Einlesen der csv Daten aus dem Header als Data-Frame erleichtert.

```
val customSchemaFlight = StructType(
  Array
  (
    StructField("YEAR", IntegerType, true),
    StructField("QUARTER", IntegerType, true),
    StructField("MONTH", IntegerType, true),
    StructField("DAY_OF_MONTH", IntegerType, true),
    StructField("DAY_OF_WEEK", IntegerType, true),
    StructField("FL_DATE", StringType, true),
    StructField("AIRLINE_ID", IntegerType, true),
    StructField("ORIGIN_AIRPORT_ID", IntegerType, true),
    StructField("ORIGIN_STATE_ABR", StringType, true),
    StructField("DEST_AIRPORT_ID", IntegerType, true),
    StructField("CRS_DEP_TIME", IntegerType, true),
    StructField("DEP_DELAY", FloatType, true),
    StructField("DEP_DELAY_NEW", FloatType, true),
    StructField("CRS_ARR_TIME", IntegerType, true),
    StructField("CANCELLED", FloatType, true),
    StructField("DISTANCE", FloatType, true),
    StructField("DISTANCE_GROUP", IntegerType, true)
  )
)

//create DataFrame from csv
val dataFrame: sql.DataFrame = sparkSession.sqlContext.read
  .format("com.databricks.spark.csv")
  .schema(customSchemaFlight)
  .option("header", "true")
  .option("delimiter", ",")
  .option("escape", "\"")
  .load("./dataFiles/FlightData17_01-02-03.csv")
```

Code 7 - Schema für Flugverkehr Statistik Daten

1.3.4 Wetter-Statistik-Daten

Die Wetter-Statistik-Daten wurden vom National Centers for Environmental Information https://www.ncdc.noaa.gov/cdo-web/search im csv Format bezogen. Analog zum Einlesen der Flugverkehr-Daten, wurde ein Custom Schema erstellt, um die Daten als DataFrame in den Spark Context einzulesen.

```
val customSchemaWeather = StructType(
  Array
  (
    StructField("STATION", StringType, true),
    StructField("NAME", StringType, true),
    StructField("DATE", StringType, true),
    StructField("PRCP", FloatType, true),
    StructField("SNOW", FloatType, true),
    StructField("SNWD", FloatType, true),
    StructField("TAVG", FloatType, true),
    StructField("TMAX", FloatType, true),
    StructField("TMIN", FloatType, true),
    StructField("WESF", FloatType, true),
    StructField("WT01", IntegerType, true),
    StructField("WT02", IntegerType, true),
    StructField("WT03", IntegerType, true),
    StructField("WT04", IntegerType, true),
    StructField("WT05", IntegerType, true),
    StructField("WT06", IntegerType, true),
    StructField("WT07", IntegerType, true),
    StructField("WT08", IntegerType, true),
    StructField("WT11", IntegerType, true)
  )
)

val weatherFrame: sql.DataFrame = sparkSession.sqlContext.read
  .format("com.databricks.spark.csv")
  .schema(customSchemaWeather)
  .option("header", "true")
  .option("delimiter", ",")
  .option("escape", "\"")
  .load("./dataFiles/WeatherData.csv")
```

Code 8 - Schema für Wetter Statistik Daten

Dieser Datensatz beschränkt sich ebenfalls ausschließlich auf Einträge für Kalifornien, USA im Zeitraum von Jänner 2017 bis März 2017.

1.3.5 Lessons Learned

Die Datenbeschaffung bzw. Datenhaltung innerhalb eines Projektes wie diesem, ist in der Regel ein langwieriger und komplexer Bestandteil dessen, und wirkt sich direkt auf die Ergebnisqualität aus. Neben der Fähigkeit des Anwenders, mit den von Spark bereitgestellten Algorithmen umzugehen, ist die Datenbeschaffung und -haltung damit der wichtigste Bestandteil der Arbeit eines Datenanalysten.
Die Medientransparenzdaten sind uns von unserem Betreuer, Herrn Dipl.-Ing. Dr.techn. Peter Salhofer zur Verfügung gestellt worden, womit die Arbeit der Datenbeschaffung aber noch nicht abgeschlossen war. Um eine Korrelation zwischen Wahlen und Ausgaben zu beobachten, haben wir einen Datensatz angelegt, der das Quartal, die Art der Wahl und das Bundesland beinhaltet (AustrianElections.csv). Dieser Datensatz wurde manuell aus öffentlichen Quellen erstellt. Für die überschaubare Größe des Datensatz (27 Zeilen) war es nicht notwendig, ein

eigenes Einlese-Programm zu schreiben. Genau dies könnte jedoch bei öffentlichen, aber nicht durch eine API, oder andere, ähnlich einfach einlesbare Daten notwendig sein – ein Beispiel wäre das Fernsehprogramm von einer öffentlichen Webseite einzuparsen. Weiters mussten die Ministerien einer politischen Orientierung zugeordnet werden, dies passiert über einen Mapper zur Laufzeit (siehe Praktische Anwendung – Medientransparenzdaten).

Auf den Flugdatensatz des United States Department of Transportation wurden wir von unserem Betreuer aufmerksam gemacht. Unser Ziel war es, diese Daten mit Wetterdaten zu verknüpfen, um eine bessere Aussage über die etwaige Verspätung von Flügen treffen zu können. Dazu haben wir mit Hilfe von Google nach US-amerikanischen Wetterdaten gesucht und mehrere Webseiten gefunden, welche für uns relevante Daten anboten. Unserem Kriterium, nach möglichst umfangreichen Daten, welche kostenlos angeboten werden, wurde schließlich die „National Oceanic and Atmospheric Administration“ gerecht. Im nächsten Schritt musste der NOAA Datensatz mit dem Datensatz des Department of Transportation verknüpft werden.
Der NBA Datensatz liegt nicht lokal vor, sondern wird zur Laufzeit mittels einer API live eingelesen. Dies erlaubt immer mit einem aktuellen Datensatz zu arbeiten.

Ein Datenanalyst muss also in der Lage sein, Daten aus unterschiedlichen Quellen zu finden, diese nach ihrer Nützlichkeit zu bewerten, diese ebenso zu beschaffen, und so aufzubereiten, dass sie gemeinsam mit anderen Datensätzen verwendet werden können.
Dass dies für Anwender, welche sich frisch mit der Materie beschäftigen, einige Fallstricke mit sich bringen kann, haben wir selbst erfahren. So haben wir für die Medientransparenzdaten bei der Linearen Regression zuerst manuell einen Custommapper geschrieben:

```
def mappData(line: String): MediaDataRow =
{
  try
  {
    //val fields = line.split(',')
    val fields = line.split(",(?=(?:[^\"]*\"[^\"]*\")*[^\"]*$)")
    MediaDataRow(fields(0), fields(1).toInt, fields(2).toInt, fields(3).toInt,
      Try { fields(4) }.getOrElse(""),        //handle null fields
      Try { fields(5).toFloat }.getOrElse(0)) //handle null fields
  }
  catch { case e: Exception => throw new Exception(s"\nError @ mapping data
lines. \nCorrupt data in line:    $line")}
}
```

Code 9 - Manueller Mapper

Später fanden wir heraus, dass die Parallelize Funktion des Spark Context uns genau diese Arbeit für CSV Dateien abnimmt.

Generell haben wir die Erfahrung gemacht, dass wir anfangs viel Zeit in das Einlesen und Aufbereiten der Datensätze investiert haben, während wir in weiterer Folge den Großteil der Zeit mit der Anwendung der Algorithmen und der Verbesserung der Resultate verbraucht haben.

2 LINEAR REGRESSION

2.1 EINLEITUNG

Das Ziel der Linearen Regression ist es, einen linearen Zusammenhang gegebener Datenpunkte zu ermitteln. Für einen 2D Plot, welcher die X-, und Y-Werte der Datenpunkte beinhaltet, wird dafür als Visualisierung eine Linie gezogen, deren jeweiliger Abstand zu den einzelnen Datenpunkten den kleinsten Fehler aufzeigt (Abbildung 3 - Lineare Regression).

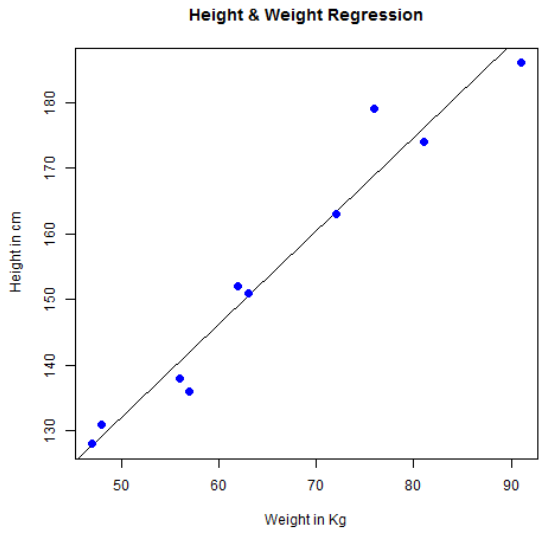

Abbildung 3 - Lineare Regression

Anhand dieser Linie kann das Modell weitere Punkte vorhersagen. Dazu ist wichtig zu spezifizieren, was der zu vorhersagende Wert sein soll. Wird für das Modell das Feld *Weight* als Label gesetzt, können für weitere Datensätze, für welche nur der Featurewert bekannt ist, der Labelwert vorhergesagt werden.

2.2 PRAKTISCHE ANWENDUNG – NBA-STATISTIK-DATEN

Dieses Experiment befasst sich mit der Vorhersage, wie viele Punkte pro Spiel ein NBA Spieler anhand eines Feature Vectors macht. Die Daten der Spieler beziehen sich auf die reguläre Saison von 2016/2017. Die Felder, welche als mögliche, aussagekräftige Features in Betracht gezogen wurden, wurden als DataFrame in Verbindung mit dem Label-Feld *pointsPG* ausgewählt.

```
val lrData = players.select(
  $"pointsPG".as("label"),
  $"position",
  $"height",
  $"weight",
  $"gamesPlayed",
  $"minSecPG",
  $"fgPct",
  $"ftPct"
)
```

Code 10 - Felder für Linear Regression

Um das Modell aufzusetzen, wurden die Features mit einer VectorAssembler Instanz für den Algorithmus vorbereitet. Die Aufteilung in Trainings- und Testdaten erfolgt zu jeweils 50%. Wie sich durch mehrere Tests und Wertebereiche der Koeffizienten zeigte, beeinflussen die Felder *fgPct*, *gamesPlayed* und *minSecPlayed* die Vorhersageergebnisse am besten, das heißt der durchschnittliche Fehler für die Vorhersagen ist in dieser Konfiguration am kleinsten.

```
//setting up features
val assembler = new VectorAssembler().setInputCols(Array(
  //"position",
  //"height",
  //"weight",
  "gamesPlayed",
  "minSecPG",
  "fgPct"
  //"ftPct"
)).setOutputCol("features")

//mapped dataset for Linear Regression
val dataLR = assembler.transform(lrData).select("label", "features")
val (trainingData, testData) = dataLR.randomSplit(Array(0.5, 0.5))
//train the lr model
val lrModel = new LinearRegression().fit(trainingData)
```

Code 11 - Setup des LR Modells

Als TestMetrik der Ergebnisse wurde der durchschnittliche Fehler der Differenz aus Label und Prediction Wert berechnet.

```
predictions.withColumn("AvgError", functions.abs($"label" - $"predic-
tion")).drop("features").describe().show()
```

Code 12 - Berechnung des durchschnittlichen Fehlers

```
+-----------------+-----------------+--------------------+--------------------+
|summary          |            label|          prediction|            AvgError|
+-----------------+-----------------+--------------------+--------------------+
|count            |              141|                 141|                 141|
|mean             |7.08226954       |  7.0914664         |   2.0157940        |
|stddev           | 5.5403879       | 5.7149065          |   1.8795687        |
|min              |              0.0|-2.0690085          |   0.0043261        |
|max              |             31.2|  17.34414          |   13.96830         |
+-----------------+-----------------+--------------------+--------------------+
```

Die *AvgError* Spalte zeigt, dass das Modell Vorhersagen mit einem durchschnittlichen Fehler von ca 2 Punkten trifft. In den folgenden Kapiteln wird die gleiche Problemstellung für den Decision Tree Algorithmus analysiert, um die Ergebnisse der verschiedenen Modelle zu vergleichen.

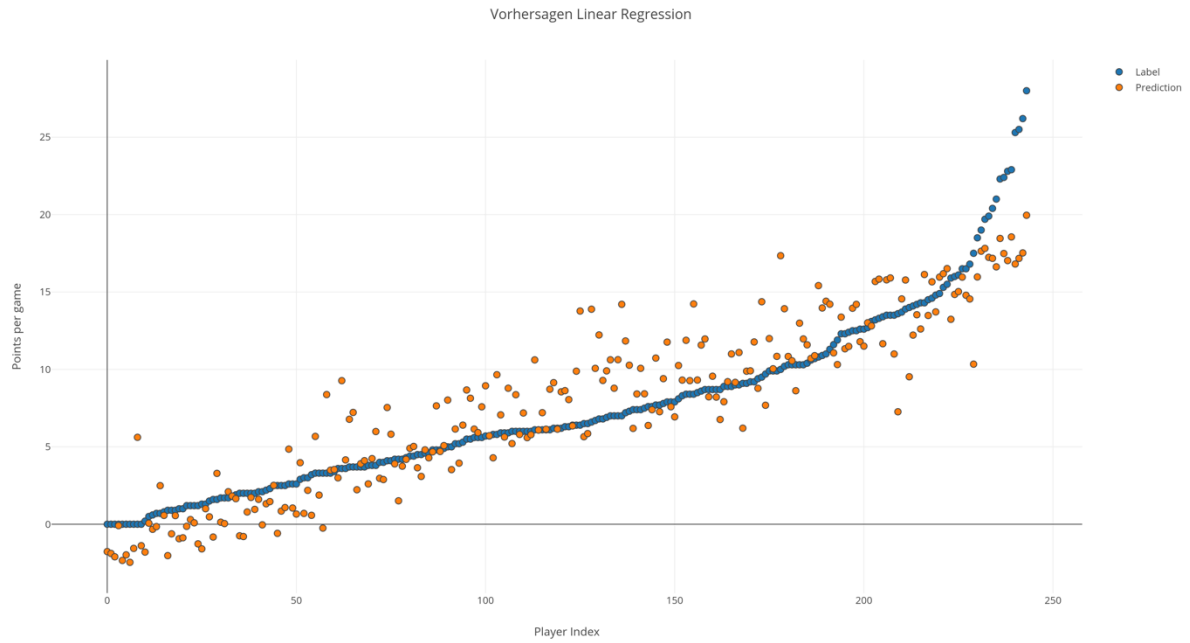

Abbildung 4 - Linear Regression Vorhersagen

Durch die entsprechende Verteilung der Prediction-Werte ist ein linearer Verlauf der Ergebnisse zu erkennen.

2.3 Lessons Learned

Die ersten Versuche dieses Projekts befassten sich mit der Linearen Regression. Dieser Umstand erforderte ein gewisses Maß an Zeit und Arbeit, diesen Algorithmus zu implementieren, obwohl die Idee dahinter immer eine sehr simple war. Angelehnt an einen empfehlenswerten Online-Kurs der Udemy Plattform (https://www.udemy.com/apache-spark-with-scala-hands-on-with-big-data/), welcher sich mit Big Data Analysen, speziell mit dem Apache Spark Framework, befasst, wurden erste Versuche gestartet, das Lineare Regressions Modell auf unsere Datensätze anzuwenden. Zu diesem Zeitpunkt war dieser Kurs erst ein Jahr alt, das Spark Framework besaß jedoch noch keine Implementierung der ML Bibliothek. Deshalb beschränkten sich unsere Versuche zu Beginn auf den MLlib Algorithmus mit der Verwendung von RDDs.

Derzeit entwickelt sich das Spark Framework jedoch immer mehr in Richtung Verwendung von DataFrames. Durch entsprechende Recherchen in diesem Bereich wurden die identen Versuche mit dem Algorithmus, der neueren ML Bibliothek, unter Verwendung von DataFrames getestet. Für uns persönlich stellte sich die Implementierung mit DataFrames als einfacher heraus, da deren Darstellung und die Syntax gegenüber RDDs leserlicher ist.

Das Einlesen der Daten wurde zu dieser Zeit häufig überarbeitet, da sich im Laufe der Zeit die Anforderungen leicht veränderten und wir auf immer bessere Techniken gestossen sind. Anfangs wurden manuelle Mapper-Funktion für das Einlesen der Daten verwendet (siehe 1.3.5).

3 LOGISTIC REGRESSION

Dieser Algorithmus kann in Spark verwendet werden, um binäre Ergebnisse vorherzusagen. Dafür muss das Label-Feld von kategorischer Natur sein. Für den Fall, dass das Label- mehr als zwei Werte bzw. Klassen hat, kann für die Analyse und Vorhersage die „Multinomial Logistic Regression“ herangezogen werden. Für die binäre Logistic Regression, versucht der Algorithmus, eine mathematische Funktion zu finden, die am besten zu dem Trainingsdatenset passt. Diese Funktion wendet die Inputdaten in Form einer Kategorisierung auf das trainierte Modell an.

3.1 PRAKTISCHE ANWENDUNG – FLUGVERKEHR-DATEN

Der Flugverkehr-Statistik-Datensatz wurde für dieses Experiment mit dem Wetterdatensatz verbunden, um die Verspätung der einzelnen Flüge vorherzusagen.

```
val flighFrame = mappedFrame.withColumnRenamed("FL_DATE", "DATE")

val weatherFrame = dataFrameMapper.weatherFrame

val joinedFrame = flighFrame.join(weatherFrame, "DATE")
```

Code 13 - Join der Datensätze auf das Datum

Als Label-Feld wurde DEPATURE_DELAY gewählt, welches in Gleitkommadarstellung im Datensatz gespeichert ist. Um den Algorithmus für binäre Logistic Regression anwenden zu können, musste im Vorhinein das Label-Feld von seinem numerischen Wertebereich in eine binäre Darstellung gebracht werden. Dafür wurde eine UserDefined-Function für das DataFrame implementiert, welche einen Wert 1 zurückgibt, wenn das Label größer als 35 Minuten ist und ansonsten 0 zurückgibt.

```
def num2bolNum: (Float => Int) = v => { if (v > 35) 1 else 0 }
```

Code 14 - Mapping des Label-Felds

Für dieses Experiment erwies sich das Festlegen der Verspätung ab 35 Minuten als „Sweetspot“ für das beste Vorhersagemodell ,mit folgenden Features aus Flug- und Wetterdaten:

```
val featureArray = Array(
  "DAY_OF_MONTH",
  "DAY_OF_WEEK",
  "AIRLINE_ID",
  "ORIGIN_AIRPORT_ID",
  "DEST_AIRPORT_ID",
  "CRS_DEP_TIME",
  "DISTANCE_GROUP",
  "PRCP",
  "SNOW",
  "SNWD",
  "TAVG",
  "TMAX",
  "TMIN",
  "WESF",
  "WT01","WT02","WT03","WT04","WT05","WT06","WT07","WT08","WT11"
)
```

Code 15 - Feature Array

Um zu bestimmen, welche Features den größten Einfluss auf die Entscheidungsfindung der Vorhersagen des Modells haben, stellt das Logistic Regression Modell die berechneten Koeffizienten aus den Features in Form eines Vektors zur Verfügung. Für eine bessere Darstellung wurden diese mit den String-Werten des Feature Arrays verknüpft und in der gewohnten, tabellarischen Ausgabeform der DataFrames, geordnet nach größtem Wert, aufgelistet.

```
val featureCoefficientMap = (featureArray zip lrModel.coefficients.toAr-
ray).map(entry => entry._1 -> entry._2).toMap

val coefficientFrame = featureCoefficientMap.toSeq.toDF("name", "value").or-
derBy($"value".desc)
```

Code 16 - Darstellung der Koeffizienten

```
Coefficients:
+-----------------+----------------------+
|name             |value                 |
+-----------------+----------------------+
|WT05             |0.2551247259204705    |
|WT03             |0.2220959824627859    |
|WT11             |0.1921894094252797    |
|WT07             |0.15132221662334358   |
|WT01             |0.1356932653339587    |
|CRS_DEP_TIME     |0.06969063988221068   |
|WT06             |0.041831478463992905  |
|TMIN             |0.016111766738196646  |
|PRCP             |0.009045661548927856  |
|TAVG             |8.005904064169473E-4  |
|SNOW             |6.871308392518637E-4  |
|AIRLINE_ID       |2.6197086912588247E-4 |
|DEST_AIRPORT_ID  |-2.4423709092840156E-6|
|SNWD             |-2.3940321886680434E-5|
|ORIGIN_AIRPORT_ID|-3.578539258983009E-5 |
|WESF             |-0.002513016619531098 |
|TMAX             |-0.011203605977443997 |
|DAY_OF_WEEK      |-0.02192375669607091  |
|DISTANCE_GROUP   |-0.026129828768076107 |
|DAY_OF_MONTH     |-0.03493974179409729  |
+-----------------+----------------------+
```

Den größten Einfluss auf Verspätungen im Modell haben demnach folgende Extremwetterdaten:

1. Hagel (WT05)
2. Gewitter (WT03)
3. Sturmböen (WT11)
4. Staub, Vulkanasche, Sandsturm, Verwehungen (WT07)
5. Nebel (WT01)

Anhand dieser Koeffizienten wurden die Vorhersagen für den Testdatensatz berechnet und die Ergebnisse ausgegeben.

```
val predictions = lrModel.transform(testData)

//show residuals
predictions.select($"label", $"prediction").describe().show()

val allCount = predictions.count()
val allLate = predictions.filter($"label" === 1).count()
val correctPredictions = predictions.filter($"label" === $"prediction").count()
val percentage = (correctPredictions / allCount) * 100

println(s"Total flights: $allCount")
println(s"Correct Predictions: $correctPredictions")
println(s"Percentage: $percentage")
```

Code 17 - Berechnung und Ausgabe der Vorhersagen

```
+-------+------------------+--------------------+
|summary|             label|          prediction|
+-------+------------------+--------------------+
|  count|         162075624|           162075624|
|   mean|0.1412417329332633|2.799248824733816...|
| stddev|0.3482707374437974| 0.01672861399698707|
|    min|                 0|                 0.0|
|    max|                 1|                 1.0|
+-------+------------------+--------------------+

Total flights: 162075624
Correct Predictions: 139166287
Percentage: 85.865032363
```

Mit dieser Konfiguration wurde die Verspätung von insgesamt 162.075.624 Flügen zu ~85,87% richtig vorhergesagt.

3.2 LESSONS LEARNED

Durch die Größe des Datensatzes und des JOINs von Wetter und Flugdaten, entstand für dieses Experiment eine viele größere Rechenlast, im Vergleich zu den anderen Anwendungen. Da ein Durchlauf auf einer Single-Client-Konfiguration bis zu einer Stunde dauerte, wurde im Projektraum der FH-JOANNEUM ein Cluster über den Spark Cluster Manager, mit drei Maschinen aufgesetzt, um die Rechenlast entsprechend aufzuteilen, bzw. tragen zu können. Dabei wurde der SparkContext mit dem Script für die Logistic Regression weiterhin über den Laptop gestartet, mit dem Unterschied, dass die Master – Worker Konfiguration über das lokale Netzwerk, auf dem Cluster Manager der Master-Maschine initialisiert wurde, welcher die Rechenlast auf seine Worker - (Slave) Maschinen aufteilte.

Der gesamte Vorgang der Konfiguration des Clusters verlief zum größten Teil problemlos. Teilweise brach der Vorgang bei wenigen Durchläufen durch eine fehlende Response einer Slave Maschine zum Master ab.

Durch den entstandenen Parallelismus des Workloads wurde, wie erwartet, eine drastische Verringerung der Rechenzeit erreicht.

Das Analysieren der aussagekräftigsten Features durch die Koeffizienten-Berechnung des Modells wurde zu Beginn vernachlässigt, was sich auf die anfänglichen Ergebnisse der Vorhersagen negativ auswirkte. In Bezug auf die Flugdaten entstand zu Beginn ein bemerklich großer Unterschied hinsichtlich unserer Erwartungen, auf den größten Einfluss einzelner Features, zu den vom Modell angegeben Koeffizienten. Die von uns erwartete Abhängigkeit der Wetterdaten wurde jedoch anhand des aufgezeigten Beispiels bestätigt und führte zu besseren Ergebnissen.

4 DECISION TREES

4.1 EINLEITUNG

Decision Trees in Machine Learning zählen zu den Supervised Learning Algorithmen. Sie eignen sich zur Lösung von Regressions-, und Klassifikationsproblemen. Analog zu dem Linear Regression Model wird durch den Decision Tree Algorithmus ein Modell erstellt, welches zuerst mit Trainingsdaten gefüttert wird. Anhand dieser Daten wird durch entsprechende Entscheidungsfindung das Verhalten des Baumes für einen folgenden Testdatensatz festgelegt. Anhand der Trainingsdaten und Entscheidungsregeln ist es dem Decision Tree möglich, die im Vorhinein festgelegten (Label) Werte für den folgenden Datensatz vorherzusagen.
Dabei besteht das Modell aus einem Label- und Feature Vektor. Je besser das Modell, im speziellen der Feature Vektor, an den verwendeten Datensatz bzw. die eigentliche Problemstellung angepasst wird, desto bessere Ergebnisse können bei der Auswertung erwartet werden.

4.2 BEISPIEL

Angenommen, es gäbe einen Datensatz für Bewerbungen einer Stelle als Software Developer in folgender Form:

Years of Experience	Level of Education	Previous Employers	Hired
5	MS	1	Yes
1	BS	0	No
10	PhD	3	Yes
3	MS	1	No
…	…	…	…

Anhand dieser Daten soll ein Decision Tree Algorithmus erstellt werden, um die Wahrscheinlichkeit zukünftiger Anstellungen anhand der vorhandenen Anstellungen vorherzusagen. Dafür wird die Spalte „Hired“ als Label gesetzt und die restlichen Spalten als Feature Vektor festgelegt. Das Decision Tree Modell aus der ***spark.mllib.tree.DecisionTree*** Library kann jedoch nur mit numerischen Datentypen Arbeiten, d.h. alle nicht numerischen Typen, wie in diesem Fall die Spalten „Level of Education“ und „Hired“ müssen dafür umgewandelt werden. Die Spalte „Hired“ kann aufgrund ihrer binären Natur einfach beispielsweise 0 für „No“ und 1 für „Yes“ gemappt werden. Die Spalte „Level of Education“ kann für dieses Beispiel mit

$$BS -> 1$$
$$MS -> 2$$
$$PhD -> 3$$

umgewandelt werden.

4.2.1.1Parameter

Für die Instanziierung des Modells sind neben den richtig formatierten Trainingsdaten weitere Parameter vonnöten, wie zum Beispiel der Parameter ***algo,*** welcher festlegt, ob es sich beim Typ des Decision Trees um Klassifikation, oder wie in diesem Beispiel, Regression handelt.
Ein weiterer wichtiger Parameter für das Modell ist ***numClasses***, dieser legt fest, wieviele Klassen es für die entsprechende Klassifikation geben soll. Für dieses Beispiel wird

$$numClasses = 2$$

gewählt, da wir uns mit „Hired“ ein binäres Ergebnis erwarten.
Die ***maxDepth*** Variable gibt an, wie viele Nodes der Tree für die Entscheidungsfindung in die Tiefe geht.

Kommen nun neue Bewerbungen zu den vorhandenen hinzu, kann das Modell mit der ***predict()*** Funktion vorhersagen, wie wahrscheinlich ein neuer Bewerber, in Bezug auf die ursprünglichen Trainingsdaten, angestellt wird.

4.3 PRAKTISCHE ANWENDUNG

4.3.1 NBA Statistik-Daten

Für die Anwendung der Decision Tree Library von Apache Spark wurden Statistik Daten von NBA Spielern mit dem Ziel verwendet, einzelne Felder wie zum Beispiel „Points per game“ anhand ausgewählter Feature-Vektoren vorherzusagen bzw. die Ergebnisse mit den tatsächlichen Daten zu vergleichen und damit die Qualität des Modells zu überprüfen. Es wurden beide Algorithmen einerseits für Spark ML, andererseits für die Spark MLlib Libraries für den gleichen Anwendungsfall getestet.

4.3.1.1 MLlib

Das Ziel dieses Experimentes war es, die Points per game für NBA Spieler mittels dem Decision Tree Algorithmus der Spark MLlib anhand folgender Features vorherzusagen:

- Games Played
- Minutes per game
- Field goal percentage
- Free throw percentage
- Position
- Weight
- Height

Der Algorithmus benötigt den Label-Feature Vector in der Form der LabeledPoint Klasse. Dafür muss zunächst das DataSet auf den Typ RDD[LabeledPoint] gemappt werden.

```
val dtData = players.map(player => LabeledPoint(
  player.pointsPG, // Get target value
  // Map feature indices to values
  Vectors.dense(
    player.gamesPlayed,
    player.minSecPG,
    player.fgPct,
    player.ftPct,
    player.position,
    player.weight,
    player.height
  )
))
```

Code 18 - Setup Feature Vector für MLlib Decision Tree

Die vorhandenen Daten wurden im 50/50 Verhältnis für Trainings- und Testdaten geteilt. In diesem Fall wurde für *impurity* „variance“ gewählt, da anstatt eines binären Ergebnis ein numerisches erwartet wurde. Für *maxdepth* und *maxBins* wurde 3 und 100 gewählt.

```
val splits = dtData.randomSplit(Array(0.5, 0.5))
val (trainingData, testData) = (splits(0), splits(1))
val categoricalFeaturesInfo = Map[Int, Int]()
val impurity = "variance"
val maxDepth = 3
val maxBins = 100

val model = DecisionTree.trainRegressor(
  trainingData,
  categoricalFeaturesInfo,
  impurity, maxDepth, maxBins
)
```

Code 19 - Decision Tree Model MLlib

Das daraus resultierende Modell kann mittels einer *model.toDebugString* Methode visualisiert werden:

```
Learned classification tree model:
DecisionTreeModel regressor of depth 3 with 15 nodes
  If (feature 1 <= 1393.0999755859375)
   If (feature 1 <= 674.0)
    If (feature 1 <= 298.6000061035156)
     Predict: 0.5157894719588129
    Else (feature 1 > 298.6000061035156)
     Predict: 2.7162161984959163
   Else (feature 1 > 674.0)
    If (feature 1 <= 1070.199951171875)
     Predict: 5.3433962363117145
    Else (feature 1 > 1070.199951171875)
     Predict: 7.7216216551290975
  Else (feature 1 > 1393.0999755859375)
   If (feature 1 <= 1920.4000244140625)
    If (feature 1 <= 1816.800048828125)
     Predict: 11.127777770713523
    Else (feature 1 > 1816.800048828125)
     Predict: 15.038461611821102
   Else (feature 1 > 1920.4000244140625)
    If (feature 3 <= 86.80000305175781)
     Predict: 20.90000009536743
    Else (feature 3 > 86.80000305175781)
     Predict: 25.014285496303014
```

Durch diese Visualisierung kann verfolgt werden, welchen Weg der Algorithmus für die jeweiligen Label Werte anhand der Features einschlägt. Erhöht man beispielsweise den Wert für *maxDepth*, erhöht sich dementsprechend die Komplexität und die möglichen Verzweigungen des Modells.
Weiters wurde der Vergleich zwischen vorhergesagtem und aktuellen Label (points per game) für die einzelnen Spieler als Spalten mit den ersten 20 Einträgen ausgegeben:

```
+----+----+
|  _1|  _2|
+----+----+
| 0.0| 0.5|
| 2.4| 5.3|
| 7.9|11.1|
| 1.9| 2.7|
| 0.0| 0.5|
| 0.9| 0.5|
|11.4|20.9|
| 4.8| 5.3|
| 3.3| 5.3|
```

```
| 2.5| 2.7|
| 5.8| 5.3|
| 8.9| 7.7|
| 6.7| 7.7|
|14.6|20.9|
| 6.6| 5.3|
| 4.7| 5.3|
| 8.7| 5.3|
|10.3|11.1|
| 1.8| 2.7|
| 2.9| 5.3|
+----+----+
```

Die Spalte _1 stellt die aktuellen Label-Werte des Datensatzes dar, _2 die vorhergesagten Werte des Models. Um eine Test-Metrik einzuführen, wurde der durchschnittliche Fehler zwischen Label und Vorhersagen berechnet:

```
val avgErr = labelAndPreds.map(r => abs(r._1 - r._2)).sum() / testData.count()
```

Code 20 - Berechnung Avg. Error der Vorhersagen

Average Error = 1.8737051792828683

Um die Vorhersagen bzw. das Model zu verbessern, kann ab diesem Punkt mit den Parametern des Models experimentiert werden. Eine Möglichkeit besteht darin, das Verhältnis der Trainings- und Testdaten zu ändern. Für dieses Beispiel konnte keine signifikante Verbesserung festgestellt werden. Eine mögliche Erklärung wäre der geringe Umfang des Datensatzes, welcher sich auf die aktuelle Anzahl der Spieler in der NBA (450 bis 500 Spieler) beschränkt.

Der Einfluss der Features auf die Ergebnisse des Models konnte in diesem Beispiel bestätigt werden. Die Felder *gamesPlayed*, *minSecPG* und *fgPct* wirken sich am stärksten auf die Ergebnisse aus. Durch mehrere Testläufe konnte eine Verbesserung des durchschnittlichen Fehlers um ~0.4 festgestellt werden.

Das Erhöhen des Parameters maxdepth führte zu einer konsistenten allgemeinen Verschlechterung der Ergebnisse.

4.3.1.2ML

Um den Unterschied zwischen den MLlib und ML Algorithmen für Decision Trees darstellen zu können, wurde derselbe Datensatz bzw. Label-Feature-Vector aus 4.3.1.1 für die ML Implementierung verwendet. Es wurde ebenfalls die gleiche Konfiguration für das Modell aufgesetzt, mit dem Unterschied, dass für die ML Implementation DataFrames anstatt RDDs vorgesehen sind. Als Vergleichsmetrik wurde, analog zum vorherigen Beispiel, der durchschnittliche Fehler der Vorhersagen berechnet.

```
val accuracy = residuals.withColumn("acurracy", functions.abs(residuals("label")
- residuals("prediction")))
```

Code 21 - Berechnung Avg. Error für ML Decision Tree

Durch die einfache Handhabung der DataFrames kann man sehr bequem eine Beschreibung des Frames mit den wichtigsten, statistischen Merkmalen ausgeben:

summary	prediction	label	accuracy
count	135	135	135
mean	7.20174	7.604444	1.610706
stddev	5.08553	5.683740	1.747629
min	0.0	0.0	0.0
max	23.5545	25.3	10.049999

Nun kann der Durchschnittswert der Accuracy mit dem Resultat der Berechnung mittels MLlib verglichen werden. Da die Trainingsdaten zufällig aus dem Datensatz gewählt wurden, wichen die Ergebnisse jeweils leicht voneinander ab. Durch mehrere Durchläufe konnte jedoch beobachtet werden, dass der Unterschied zwischen den Berechnungen der zwei Implementierungen vernachlässigbar klein ist. Um die Ergebnisse visualisieren zu können, wurde mithilfe von Plotly ein Plot der Label- und Prediction-Werte des Ergebnisses berechnet.

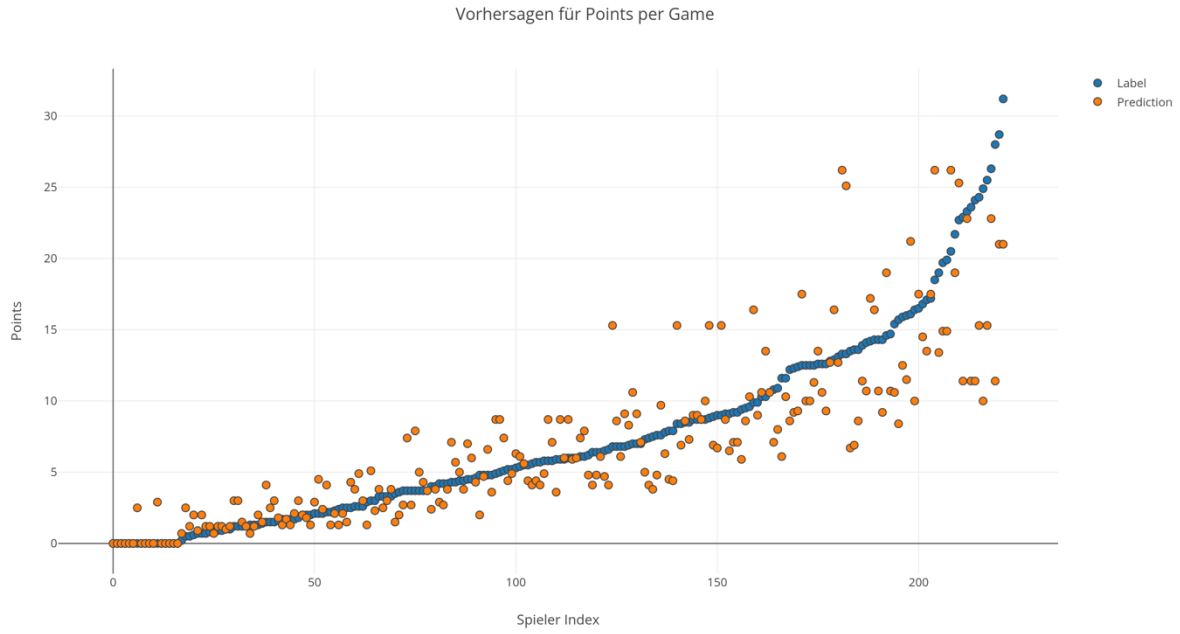

Abbildung 5 - Visualisierung: Label-Prediction

Von 0 bis ca. 10 Punkten ist eine stärkere Korrelation zu sehen, da sich der Großteil der Spieler in diesem Punktebereich pro Spiel bewegt.

Im direkten Vergleich zur Anwendung der Linearen Regression (siehe 2.2), in der die gleiche Konfiguration an Daten, Label und Feature Vektor verwendet wurde, produzierte der Decision Tree für diese Problemstellung im Durchschnitt leicht bessere Ergebnisse.

4.3.2 Flugverkehr-Statistik-Daten

Dieses Experiment befasst sich mit der Vorhersage möglicher Verspätungen von Flügen, mittels Decision Tree Algorithmus. Für folgendes Beispiel wurde der Flugdatensatz bzw. Wetterdatensatz auf Kalifornien bzw. die Zeitspanne von Jänner 2017 bis März 2017 beschränkt (siehe 1.3.3, 1.3.4).

Als Label wurde das Feld *DEP_DELAY gewählt*, welches einen Minuten-, Sekundenwert im Gleitkommaformat darstellt, auf eine binäre Darstellung gemappt. Dafür wurde eine UserDefined Function verwendet, die Flüge mit einem *DEP_DELAY* > 35 als verspätet (1) mappt und alles darunter als pünktlich (0) kennzeichnet.

```
def num2bolNum: (Float => Int) = v => if (v > 35) 1 else 0
```

Code 22 - Mapping der Label für Verspätung

Das resultierende Feld wurde als Label für das Decision Tree Modell festgelegt.
Vor dem Setup des Modells war es notwendig, die Wetter Daten mit den Flugdaten in einer Collection zu verbinden. Dafür wurde ein JOIN der beiden DataFrames anhand des *DATE* Feldes durchgeführt.

```
val completeFrame = renamedFrame.join(weatherFrame, "DATE")
```

Code 23 - Join: Flugdaten mit Wetterdaten

Um die Verspätung eines Fluges vorherzusagen, wurden folgende Felder als Features in der Form von Label -> Vector als LabeledPoint verwendet:

```
val dtData = expandedFrame.map(row => LabeledPoint(
  row.getAs[Int]("IS_DELAYED"),
  Vectors.dense(
    row.getAs[Int]("MONTH"),
    row.getAs[Int]("DAY_OF_MONTH"),
    row.getAs[Int]("DAY_OF_WEEK"),
    row.getAs[Int]("AIRLINE_ID"),
    row.getAs[Int]("ORIGIN_AIRPORT_ID"),
    row.getAs[Int]("DEST_AIRPORT_ID"),
    row.getAs[Int]("CRS_DEP_TIME"),
    row.getAs[Int]("DISTANCE_GROUP"),
    row.getAs[Float]("PRCP"),
    row.getAs[Float]("SNOW"),
    row.getAs[Float]("SNWD"),
    row.getAs[Float]("TAVG"),
    row.getAs[Float]("TMAX"),
    row.getAs[Float]("TMIN"),
    row.getAs[Float]("WESF"),
    row.getAs[Int]("WT01"),
    row.getAs[Int]("WT02"),
    row.getAs[Int]("WT03"),
    row.getAs[Int]("WT04"),
    row.getAs[Int]("WT05"),
    row.getAs[Int]("WT06"),
    row.getAs[Int]("WT07"),
    row.getAs[Int]("WT08"),
    row.getAs[Int]("WT11")
  )))
```

Code 24 - Label-Features für Flug- und Wetterdaten

Da das Label nun einen binären Wertebereich darstellt und ein binäres Ergebnis erwartet wurde, wurde für das Modell der Parameter *numClasses* = 2 und als Methode *trainClassifier* verwendet.

```
//split data for training and testing
val splits = dtData.randomSplit(Array(0.5, 0.5))
val (trainingData, testData) = (splits(0), splits(1))

val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
val impurity = "gini"
val maxDepth = 5
val maxBins = 32

val model = DecisionTree.trainClassifier(trainingData, numClasses, categori-
calFeaturesInfo,
  impurity, maxDepth, maxBins)
```

Code 25 - Decision Tree Classifier Modell

Um die Vorhersagen zu berechnen und die Ergebnisse ausgeben zu können, wurde ein Mapping der Modellergebnisse zu einem DataFrame durchgeführt. Im Anschluss wurden Prozentwerte der korrekten Vorhersagen ausgegeben.

```
val labelAndPreds = testData.map(entry => (entry.label, model.predict(entry.fea-
tures))).toDF()
val correctCount = labelAndPreds.select("*").where("_1 = _2").count()
val percentage: Float = correctCount.asInstanceOf[Float] / testData.count() *
100

println(s"\nTotal: ${testData.count()}")
println(s"Correct predictions: $correctCount")
println(s"Percentage: $percentage")
```

Code 26 - Berechnung der Vorhersagen

```
Total:                  162075624
Correct predictions:    131063575
Percentage:             80.865692
```

Anhand der Ausgabe ist der Umfang des Datensatzes mit über 160 Millionen Zeilen durchaus im Big Data Bereich, was sich durch den Zeitaufwand der Berechnung eines Single Client zeigt. Für dieses Beispiel empfiehlt sich, die Verwendung eines Clusters zur Berechnung. Der Zeitaufwand für die Berechnung dieser Konfiguration auf der folgenden Maschine:

MacBook Pro (15-inch, 2016)
2,6 GHz Intel Core i7
16 GB 2133 MHz LPDDR3

beläuft sich auf über eine Stunde.

Aufgrund des hohen Zeitaufwandes der Berechnung, wurden weniger Testläufe für dieses Beispiel durchgeführt. Die Verwendung der Wetterdaten, zusätzlich zu den Flugdaten, führte zu einer eindeutigen Verbesserung der Vorhersagen, da ein Testlauf ohne den Wetterdatensatz zu einem korrekt vorhergesagten Prozentsatz von ~65% führte. Im Vergleich zur Logistic

Regression, mit der gleichen Konfiguration an Daten und Features, erwies sich der Decision Tree insgesamt als etwas weniger geeignet, wenn man die Prozentzahlen der richtigen Vorhersagen von

Logistic Regression: ~85%
Decision Tree: ~80%

betrachtet.

4.4 Lessons Learned

Was im ersten Moment komplex wirkte, stellte sich einfacher als erwartet heraus. Die Grundidee eines Entscheidungsbaumes ist eine sehr einfache, wenn man von der eigentlichen Implementierung des Algorithmus, auf die in dieser Arbeit nicht weiter eingegangen wird, absieht. Das größte Problem stellte erneut die Datenaufbereitung des Modells selbst dar. Nach deren Umsetzung verlief der weitere Ablauf Großteils analog zu den bereits getesteten Algorithmen. Nach der Implementierung des ersten lauffähigen Vorhersagemodells, wurde eine Vielzahl an Tests durchgeführt, mit dem Ziel die Vorhersagegenauigkeit zu verbessern.
Es konnte ein direkter Vergleich zu den Ergebnissen des gleichen Beispiels, durch die Lineare Regression hergestellt werden. Im Durchschnitt ergaben beide Algorithmen für den gleichen Datensatz mit derselben Label-Feature Konfiguration einen durchschnittlichen Fehler von ~1.8 für die Vorhersage von Punkten pro Spiel.

5 CLUSTERING

Clustering ist eine Technik von unüberwachtem Lernen. Dabei wird nicht versucht ein bestimmter Wert vorherzusagen, sondern es wird versucht, im strukturlosen Rauschen der Daten, Strukturen und natürliche Gruppierungen zu finden. Das Ziel ist es, Datenpunkte welche ähnlich sind, in ein Cluster zu klassifizieren, während andere, unähnliche Datenpunkte in andere Cluster klassifiziert werden.
Ein Beispiel für die praktische Anwendung von Clustering findet sich mithilfe der Medientransparenz-Daten. Es sollen Organisationen mit bekannter politischer Prägung untersucht werden, um eine Aussage treffen zu können, ob die politische Prägung einer Organisation deren Ausgabeverhalten beeinflusst.
Dafür müssen sogenannte Anomalien im Datensatz ausgemacht werden. Anomalien sind Faktoren, welche den Datensatz beeinflussen, aber noch nicht bekannt sind. Werden diese Faktoren gefunden und verstanden, handelt es sich nicht länger um Anomalien.

5.1 K-MEANS CLUSTERING

„K-means“ ist ein verbreiteter Algorithmus im Umgang mit Clustern. Der Algorithmus versucht k Cluster auszumachen, wobei k eine natürliche Zahl größer Null ist, welche vom Anwender bestimmt wird. Jeder Datensatz erfordert einen anderen Wert für k, wobei dieser gewählte Wert einen direkten Einfluss auf die Qualität des Resultats hat. Der ideale Wert für k lässt sich nur durch Probieren mehrerer Werte finden – dazu mehr im Abschnitt „Praktische Anwendung“.
Der K-means Algorithmus kann nur mit numerischen Werten umgehen, um mit nicht-numerischen Daten, wie Strings (z.B. Organisationsname) arbeiten zu können, müssen diese auf numerische Werte gemappt werden.
Ein Cluster wird bei der Anwendung des K-means Algorithmus durch einen Punkt definiert. Dieser Clustercenter ist der Mittelpunkt aller Datenpunkte, welche dem Cluster angehören, und wird durch den Mittelwert aller jener Datenpunkte bestimmt. Daher stammt auch der Name des Algorithmus – „means“ dies bedeutet im Englischen, auf die Mathematik bezogen „Mittelwert“.
Im ersten Schritt werden k Clustercenter gesetzt. Gibt der Anwender keine Daten (sogenannte „Seeds“ – englisch für Samen) mit, werden dazu k Datenpunkte aus dem Datensatz zufällig gewählt. Dies geschieht, weil vor dem Ausführen des K-means Algorithmus die idealen Clustercenter noch nicht bekannt sind, hat aber den Nebeneffekt, dass bei jeder Ausführung des Algorithmus ein unterschiedliches Resultat berechnet werden kann. Dafür sind lokale Minima verantwortlich, welche eine ideale, globale Lösung verhindern.
Im bebilderten Beispiel wurde k auf 3 festgelegt. Die grauen Rechtecke stellen Datenpunkte des verwendeten Datensatzes dar. Die anfänglichen Clustercentren wurden bestimmt.

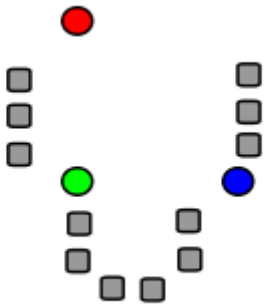

Abbildung 6 - Kmeans, anfängliche Clustercentren

Im nächsten Schritt werden die Datenpunkte dem Clustercenter mit dem geringsten Abstand zugeordnet.

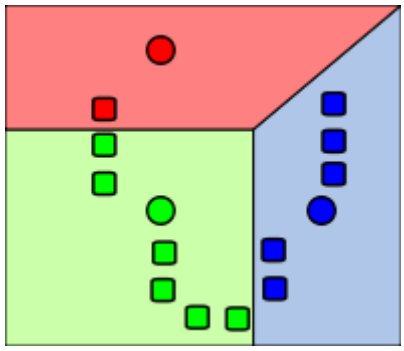

Abbildung 7 - Kmeans, Zuordnung der Datenpunkte

Um den Abstand zwischen den Datenpunkten zu berechnen, wird deren euklidischer Abstand zum Clustercenter berechnet. Das heißt, im zweidimensionalen Raum, dass die Distanz zwischen zwei Datenpunkten mittels Satz des Pythagoras ermittelt wird, wobei die Differenz der x-Koordinaten die erste Kathete und die Differenz der y-Koordinaten die zweite Kathete darstellt. Die Distanz ist folglich der Betrag der Hypotenuse. Der euklidische Abstand verwendet den Satz des Pythagoras und erlaubt Berechnungen im n-dimensionalen Raum.
Ähnliche Datenpunkte haben eine geringe Distanz, während unähnliche Datenpunkten eine große Distanz aufweisen. Dabei sollte unbedingt beachtet werden, dass es sich hierbei um relative Werte handelt und deren Betrag sich von Datensatz zu Datensatz unterscheidet.
Im dritten Schritt werden die Clustercentren neu gesetzt, in dem der Mittelwert aller dem Cluster zugehörigen Datenpunkte berechnet wird.

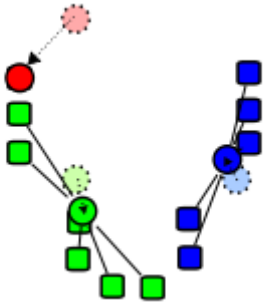

Abbildung 8 - Kmeans, Versetzung der Clustercentren

Im vierten Schritt wird die Zuordnung der Datenpunkte zu ihrem Cluster aufgehoben. Anschließend wird für jeden Datenpunkt erneut das Clustercenter mit dem geringsten Abstand

berechnet und dessen Cluster zugewiesen. Dies hat zur Folge, dass die Clusterzugehörigkeit von Datenpunkten sich in diesem Schritt ändern kann.

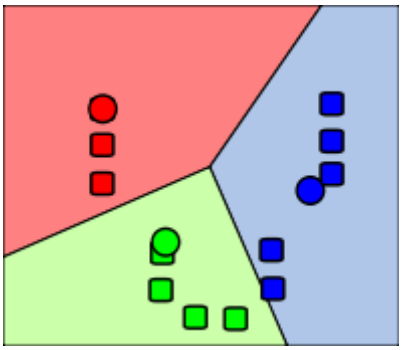

Abbildung 9 - Kmeans, erneute Zuordnung der Datenpunkte

Schritt 3 und 4 werden daraufhin mehrfach wiederholt. Wie oft hängt davon ab, ob der Anwender eine Anzahl an Durchläufen spezifiziert hat, ansonsten greift der hinterlegte Standardwert (der Standardwert legt 20 Durchläufe fest) (vgl. [Ryza, Laserson, Owen, Wills 2017], S. 105). Die Schleife wird auch dann unterbrochen, wenn zwischen den Iterationen keine Änderung mehr stattfindet – also eine ideale Lösung für jene Clustercenter gefunden wurde, welche Anfangs bestimmt wurden.

5.2 Praktische Anwendung – Medientransparenz Daten

Für die praktische Anwendung wurde die ML Bibliothek von Spark verwendet. Es wurde das Ausgabeverhalten von politischen Organisationen untersucht. Zu diesem Zweck wurden im ersten Schritt aus dem Medientransparenz Datensatz die Ausgaben aller Ministerien und des Bundeskanzleramts herausgefiltert. Weiters wurden nur Zahlungen ab Quartal 1 2014 berücksichtigt (Beginn der Amtszeit der Regierung Faymann II: 16. Dezember 2013) und das Quartal 2 2016 wurde herausgefiltert, da am 17. Mai 2016 die Regierung Kern ihre Arbeit aufnahm (vgl. [https://de.wikipedia.org][2]).

```
val filteredFrame = dataFrame
  .filter(dataFrame("organisation").contains("ministerium") || dataFrame("organ-
isation").contains("kanzleramt"))
  .filter(dataFrame("period") >= 20141)   //start for Faymann
  .filter(dataFrame("period") =!= 20162) //unsure of influence -> 17.05.2016
Faymann handing over to Kern
```

Code 27 - Filtern des Medien DataFrames

Somit bleiben nur noch Organisationen mit bekannter politischer Prägung. Mit Prägung oder Orientierung ist gemeint, welche Partei den jeweiligen Minister oder die jeweilige Ministerin des Ministeriums stellt. Interessanterweise haben alle Ministerien von denen Zahlungen vorliegen, in den Regierungsperioden Faymann II und Kern ihre politische Orientierung beibehalten. Die politische Orientierung der Ministerien wurde den folgenden Artikeln entnommen: https://de.wikipedia.org/wiki/Bundesregierung_Faymann_II#%C3%9Cbersicht und https://de.wikipedia.org/wiki/Bundesregierung_Kern#Regierungsmitglieder .
Um die Zahlungen des Bundeskanzleramtes und der Ministerien im Datensatz mit der jeweiligen politischen Orientierung. Diese wurden folgendermaßen zugeordnet:

Zuordnung	Politische Orientierung	Anmerkung
1	ÖVP	-
2	SPÖ	-
3	Beide	Ministerium/Amt das von beiden Parteien beeinflusst wird (nur Bundeskanzleramt)
9	Andere	Ministerium/Amt das nicht zugeordnet werden kann (nur Familienministerium)

Tabelle 2 - Zuordnung politische Orientierung

Diese Zuordnung erfolgte über folgenden Code:

```
def orgToPolOrient = udf((orgName: String) =>
{
  if (orgName == "Bundeskanzleramt") 3
  else if (orgName.contains("Arbeit")) 2
  else if (orgName.contains("Bildung")) 2
  else if (orgName.contains("Europa")) 1
  else if (orgName.contains("Familie")) 9
  else if (orgName.contains("Finanzen")) 1
  else if (orgName.contains("Gesundheit")) 2
  else if (orgName.contains("Inneres")) 1
  else if (orgName.contains("Umwelt")) 1
  else if (orgName.contains("Landesverteidigung")) 2
  else if (orgName.contains("Verkehr")) 2
  else if (orgName.contains("Wissenschaft")) 1
  else 999  //dummy for other if filter went wrong
})
```

Code 28 - Mapping der Organisation auf politsche Orientierung

Wie beschrieben, kann der K-means Algorithmus nur mit numerischen Werten umgehen. Daher ist es notwendig, dass auch Organisations- und Mediennamen auf numerische Werte gemappt werden.

```
//organisation key value map for string to numeric conversion
val organisationColumn = polOrientFrame.select("organisation").groupBy("organi-
sation").count().drop("count")
val organisationMap = organisationColumn.rdd.zipWithIndex.map(row =>
(row._1(0).asInstanceOf[String], row._2)).collectAsMap()

//media key value map for string to numeric conversion
val mediaColumn = polOrientFrame.select("media").groupBy("me-
dia").count().drop("count")
val mediaMap = mediaColumn.rdd.zipWithIndex.map(row => (row._1(0).asIn-
stanceOf[String], row._2)).collectAsMap()

//UDFs for assigning index key to org/media description
def organisationToIndex: (String => Long) = v => organisationMap.getOrElse(v, -
1)
def mediaToIndex: (String => Long) = v => mediaMap.getOrElse(v, -1)
//def federalStateToInt: (String => Int) = v => v.substring(3).toInt

val organisationToIndex_udf = udf(organisationToIndex)
val mediaToIndex_udf = udf(mediaToIndex)
//val federalStateToInt_udf = udf(federalStateToInt)

//setup of numeric frame for clustering algorithm
val indexedFrame = polOrientFrame
  .withColumn("organisation", organisationToIndex_udf(dataFrame("organisa-
tion")))
  .withColumn("media", mediaToIndex_udf(dataFrame("media")))
```

Code 29 – Mapping: String -> Numeric

In weiterer Folge kann das Model für den K-means Algorithmus erstellt werden. Dies geschieht in der „computeCost“ Funktion.

```
def computeCost(data: DataFrame, k: Int): Double =
{
  val assembler = new VectorAssembler().setInputCols(data.col-
umns).setOutputCol("features")

  val kMeans = new KMeans()
    .setSeed(Random.nextLong())
    .setK(k)
    .setMaxIter(40)
    .setTol(1.0e-5)
    .setPredictionCol("cluster")
    .setFeaturesCol("features")

  val pipeline = new Pipeline().setStages(Array(assembler, kMeans))

  val kMeansModel = pipeline.fit(data).stages.last.asInstanceOf[KMeansModel]

  kMeansModel.computeCost(assembler.transform(data)) / data.count()
}
```

Code 30 - KMeans Modell in computeCost Funktion

Das KMeans Modell benötigt zur Auswertung des vorbereiteten DataFrames ein Pipeline Object, welches wiederum abhängig von einem VectorAssembler Object ist. Letzteres wird von der Pipeline benötigt, um festzulegen, welche Felder bzw. Columns im DataFrame als Features herangezogen werden sollen.
Mit setSeed werden die anfänglichen Clustercentren gewählt. Da bei unserem Beispiel keine bekannten Centren vorlagen, wurden diese dem Zufall überlassen. Mittels setK wird die Anzahl an Clustercentren festgelegt. Da die ideale Anzahl erst bestimmt werden musste, haben wir k als Parameter der Methode festgelegt um die Methode mit unterschiedlichen Werten für k aufrufen zu können. Die setMaxIter Funktion legt die maximale Anzahl an Durchläufe fest. Default ist 20, eine Erhöhung auf 40 brachte eine erhöhte Genauigkeit auf Kosten der Rechenzeit mit sich. Eine weitere Erhöhung zeigte keine Verbesserung der Genauigkeit. Mittels setTol wird der Grenzwert festgelegt ab wann eine Clustercenter Verlegung als nicht mehr signifikant betrachtet wird und auf weitere Durchläufe verzichtet wird.
Der Score, welcher mittels der Methode computeCost bestimmt wird, ist die durchschnittliche euklidische Distanz der Datenpunkte zu ihrem jeweiligen Clustercenter und gibt somit die Genauigkeit an. Dementsprechend kann nun mittels folgenden Aufruf die Genauigkeit des K-means Algorithmus für eine Spannweite von Werten bestimmt werden.

```
val kList = (2 to 16 by 2 ).map(k => computeCost(numericFrame, k))
```

Code 31 - Berechnung der Genauigkeit

Dadurch erhält man folgende Ergebnisse:

Anzahl der Cluster	Score
2	2.964289440689424E12
4	6.800880605896903E11
6	1.3229474243709906E11
8	6.839057403277546E10
10	3.9978831474381E10
12	2.400670905190382E10
14	1.7878521428890602E10
16	1.145049080974914E10

Tabelle 3 - Clusterzuordnung

Mit Hilfe von Plotly wurde folgender Graph erstellt:

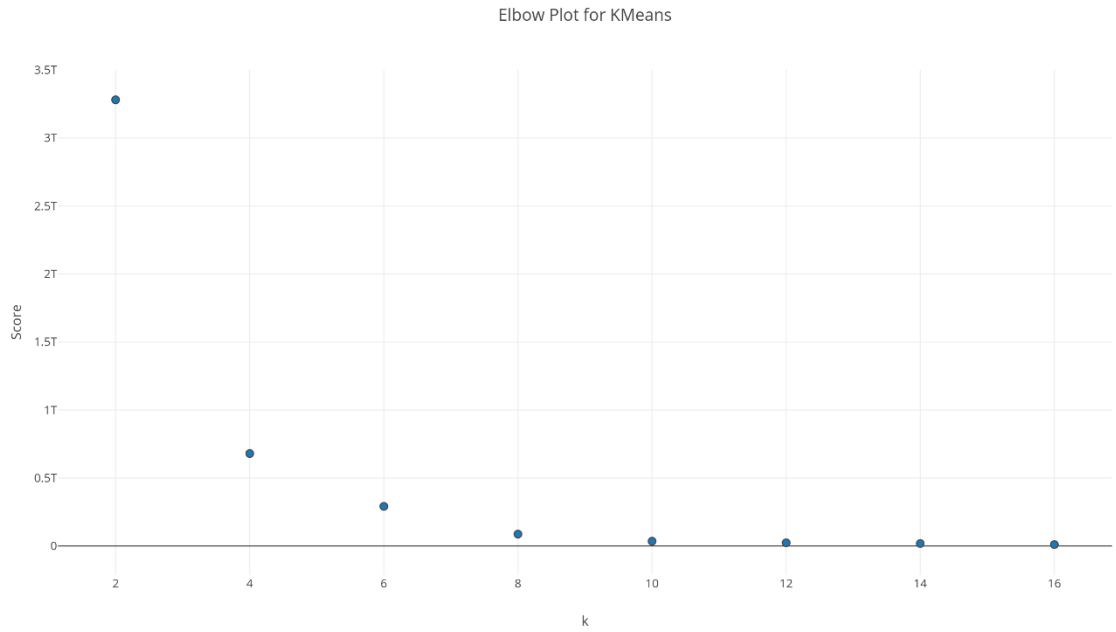

Abbildung 10 – Clusterzuordnung

Wie beschrieben ist der Score mit der Genauigkeit der Clusterzuordnung gleichzusetzen. Desto mehr Cluster es gibt, desto näher sind die Datenpunkte im Durchschnitt an dem nächsten Clustercenter. Die Genauigkeit nähert sich immer weiter dem Quasiidealwert 0 an, welcher erreicht wird sobald es so viele Clustercentren wie Datenpunkte gibt. Dies ist natürlich nicht das Ziel des Clusterings also muss immer ein gewisser Fehler akzeptiert werden. Wo dieser Wert liegt ist wiederum vom Datensatz abhängig.
Dazu wird der sogenannte „Elbow“ (engl. für Ellbogen) im Graphen gesucht – das ist jene Stelle an der, der Genauigkeitsgewinn abflacht. In unserem Beispiel mit den Medientransparenzdaten ist dies bei k = 4. Hat man sich auf den Elbow festgelegt, erhöht man k zur Sicherheit auf den nächsthöheren Wert. Dies ist das optimale k für den jeweiligen Datensatz, für uns also 6.

```
val withCluster = pipelineModel.transform(numericFrame)

val wCluster = withCluster.select("cluster", "polOr").groupBy("cluster",
"polOr").count()

wCluster.orderBy(wCluster("cluster"), wCluster("count").desc).show()
```

Code 32 - Berechnungund Ausgabe der Cluster des KMeans Algorithmus

Die Zuordnung der Clustercentren anhand des Datensatzes (numericFrame), wurde durch das Pipeline Modell umgesetzt. Danach wurde das entstandene DataFrame in eine leserliche Form für die Ausgabe gemappt.

cluster	polOr	count
0	1 (ÖVP)	185
0	2 (SPÖ)	93
1	2 (SPÖ)	103
2	9 (Other)	77
2	1 (ÖVP)	51
3	2 (SPÖ)	87
3	1 (ÖVP)	14
4	2 (SPÖ)	106
4	1 (ÖVP)	103
5	3 (Both)	67

Tabelle 4 – Clusterzuordnung

Ergebnis von k-means Algorithmus bei k = 6.
Interpretation: Cluster #0, #3 und #4 enthalten sowohl ÖVP als auch SPÖ Ministerien. Diese Cluster erlauben keinen Rückschluss auf ein differenziertes Ausgabeverhalten. Cluster #1 und #5 enthalten jeweils eine politische Orientierung – Cluster 1 ist eindeutig der SPÖ zuzuordnen was auf ein Ausgabeverhalten schließen lässt, welches abweichend der anderen Zahlung ist. Eine genauere Untersuchung dieses Clusters könnte weiteren Aufschluss liefern. Cluster #5 ist eindeutig dem Bundeskanzleramt zuzuordnen, dessen Ausgabeverhalten von jenem der Ministerien abgrenzbar ist.

Am interessantesten ist Cluster #2, welches die politische Orientierung 9 enthält. Diese Klassifizierung trifft nur auf das Familien- und Jugendministerium zu. Sowohl unter der Regierung Faymann II, als auch unter der Regierung Kern war dieses Ministerium unter der Leitung von Ministerin Sophie Karmasin. Diese wurde von der ÖVP nominiert, gilt allerdings als parteilos. Cluster #2 enthält zudem Zahlungen von einem oder mehreren ÖVP Ministerien wodurch der Schluss gezogen werden kann, dass das Familienministerium trotz der offiziellen Parteilosigkeit der Ministerin ein Ausgabeverhalten aufweist welches ÖVP-nahe oder -geprägt zu sein scheint.

5.3 LESSONS LEARNED / VERFOLGTE ANSÄTZE

Beim Clustering handelt es sich um ein Problem des "Unsupervised Learnings", es sollen wie beschrieben Anomalien im Datensatz gefunden werden. Dies erwies sich im Vergleich zu den anderen Bereichen dieser Arbeit insofern als Problem als dass kein klares Ziel definiert war, wie eben beim Flugdatensatz, bei dem Aussagen betreffend der Verspätung der Flüge getroffen werden sollten. Uns war bekannt, dass wir mit dem K-means Algorithmus arbeiten sollten, und haben dazu ein halbes Duzend Tutorials im Internet durchgelesen welche aus unterschiedlichen Gründen nicht auf unseren Datensatz anwendbar waren. Zu diesem Zeitpunkt hatten wir bereits einige Ansätze probiert, welche teilweise im Commitment-Verlauf unseres Repositories zu finden sind. Wir erkannten, dass der im Advanced Analytics with Spark-Buch beschriebene Weg, der für uns am vielversprechendste war und konnten diesen auf unseren Datensatz adaptieren. Eine wichtige Erkenntnis war, dass die Herangehensweise an das Clustering sich durchaus nach Datensatz unterscheidet und wie man vorhandene Algorithmen und Ansätze an die individuellen Features eines Datensatzes anpasst, beziehungsweise den Datensatz auf den Algorithmus anpasst (Mapping von nicht-numerischen Werten).

Diese beschriebenen Probleme waren auf Basis von mangelndem Verständnis von Clustering und teilweise programmatischer Natur. Ein weiters Problemfeld waren logische Probleme, welche für Einsteiger in die Materie nur schwer erkennbar sind und immer wieder in Sackgassen mit falschen oder unlogischen Ergebnissen geführt haben. So war es notwendig die Ausgaben jeder Organisation zu normalisieren um mit dem prozentuellen Anteil der Ausgaben an die Medien arbeiten zu können. Dabei haben wir Anfangs die Summen normalisiert, welche an die Medien gezahlt wurden, was jedoch hieß, dass eine Organisation deutlich mehr oder weniger als 100% ausgeben konnte. Dieser und andere Logikfehler haben uns beim Clustering viel Zeit gekostet und wären ohne die Hilfe unseres Betreuers Professor Salhofer fatal für diesen Abschnitt geworden, welcher uns immer wieder zurück auf Spur brachte.

Die zeitaufwändigste Schwierigkeit dieses Experimentes war die Aufbereitung des Datensatzes, die insgesamt die meisten Arbeitsstunden des gesamten Projektes beanspruchte. Die ersten, mit RDDs implementierten Ansätze erwiesen sich für die verfolgte Problemstellung als ungeeignet. Der erste teilerfolgreiche Ansatz wurde mit einer DataFrame basierten Pivot-Table Berechnung umgesetzt.

```
val pivotTable = indexedFrame
  .groupBy("organisation")
  .pivot("media")                  //pivot element for aggregate function
  .agg(sum(indexedFrame("%")))     //aggregate with percentage of spendings by org
  .sort("organisation")
  .na.fill(0)                      //replace null values with 0
```

Code 33 - Berechnung Pivot Tabelle

Die Berechnung führte zu folgendem DataFrame:

Organisation	Media 1	Media 2	Media 3	Media 4	…..
0	0	0	0	0	
0	0	0	0	0	
1	0	2.321	0	0	
2	0	0	0	0	
2	0.342	0	0	0	
2	0	0	0	0	
3	0	0	0	0	
4	0	0	0	0	
4	0	0	0	0	
5	0	0.045	0	0	

Tabelle 5 - Pivot Tabelle

Als Spalteneinträge wurden die Zahlungen der Organisationen an die jeweiligen Medien in Prozent berechnet. Das Problem dieser Berechnung war die folgende Darstellung und Interpretation des Ergebnisses durch K-Means, da es uns nicht möglich war durch diese Aufbereitung auf politische Zusammenhänge rückzuschließen. Daraufhin wurde ein Ansatz ausgearbeitet, welcher einem Beispiel aus (vgl. [Ryza, Laserson, Owen, Wills 2017], S. 97ff) gleicht. Dieser verwendete ein Pipeline Objekt für die Aufbereitung der Daten als Alternative zu unserer manuellen Methode. Dadurch ermöglichte sich eine einfachere Implementierung und Auswertung der Cluster-Ergebnisse für unser Beispiel.

6 RESÜMEE

Spark stellt eine Vielzahl an Algorithmen für die verschiedensten Maschine Learning Anwendungen und Problemstellungen zur Verfügung. Es liegt alleine am Data Scientist zu bestimmen, welches Modell die Ansprüche seiner Problemstellung am besten unterstützt. Wo anfangs ein einzelner Hammer in der Toolbox verfügbar ist, sammeln sich durch Weiterbildung und Erfahrung immer mehr Werkzeuge in der Toolbox an, um eine gute Antwort auf jede Problemstellung zu haben. Durch unsere intensive Beschäftigung mit der Thematik, ist unsere Toolbox für den Bereich Data Science sicherlich weitergewachsen.

Während wir zu Beginn, teilweise aber auch noch in weiterer Folge unserer Arbeit von der enormen Vielfalt und Mächtigkeit des Frameworks überwältigt waren, wich diese Unsicherheit spätestens dann Enthusiasmus, als wir erste plausible Ergebnisse aus einem Datensatz mit Millionen von Datenpunkten gewinnen konnten. Es kann durchaus von einer Art Befriedigung gesprochen werden, wenn man aus dem Chaos einer solch enormen Datenmasse, erfolgreich brauchbare Informationen abstrahieren kann.
Wir beide wollen nach dem erfolgreichen Abschluss unseres Bachelorstudienganges, den Masterstudiengang „Data and Information Science“ absolvieren, weshalb wir bereits vor dieser Arbeit mit Thematiken wie Big Data und Machine Learning vertraut waren. Diese Arbeit hat unser Wissen in diesen Bereichen vertieft und uns in unserer Entscheidung, besagtes Masterstudium anhängen zu wollen, bestärkt.
Sind wir nun Datenanlysten? Nein. Sind wir einen Schritt näher daran Datenanalysten zu sein? Ja.

LITERATURVERZEICHNIS

6.1.1.1Bücher

[Ryza, Laserson, Owen, Wills 2017]
Ryza S., Laserson U., Owen S., Wills J.: Advanced Analytics with Spark. O'Reilly Verlag 2017
[**Guller 2015**]
Guller M.: Big Data Analytics with Spark - A Practitioner's Guide to Using Spark for Large Scale Data Analysis. Apress Verlag 2015

6.1.1.2Internet-Referenzen

[https://de.wikipedia.org]
[1] Apache Spark; https://de.wikipedia.org/wiki/Apache_Spark; 26.02.2018
[2] Österreichische Bundesregierungen in der Zweiten Republik; https://de.wikipedia.org/wiki/Bundesregierung_(%C3%96sterreich)#Zweite_Republik_(seit_1945); 15.02.2018

[https://spark.apache.org]
[1] Apache Spark Documentation; https://spark.apache.org/docs/latest/ml-guide.html; 03.03.2018