Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Jannik Bruhns

Machine Learning im Big Data Umfeld – Experimenteller Vergleich von Apache Flink und Apache Spark

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Jannik Bruhns

Machine Learning im Big Data Umfeld – Experimenteller Vergleich von Apache Flink und Apache Spark

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 25. Juni 2019

Jannik Bruhns

Thema der Arbeit

Machine Learning im Big Data Umfeld – Experimenteller Vergleich von Apache Flink und Apache Spark

Stichworte

Apache Flink, Apache Spark, Machine Learning, MLlib, Spark ML, Flink ML, Big Data, Experimente, Support Vector Machine, MinMax Scaler, K-Means, Stochastic Outlier Selection, Multiple Linear Regression, Alternating Least Squares, Implementierung, Vergleich

Kurzzusammenfassung

Das Große Thema in der Informatik ist zur Zeit der Bereich der „Künstlichen Intelligenz“, ein Teilbereich aus diesem ist das „Maschinelle Lernen“. Die Open Source Big Data Frameworks Apache Flink und Apache Spark bieten verschiedene Bibliotheken und Schnittstellen an, unter anderem für Machine Learning. In dieser Thesis soll herausgefunden werden, welches Framework sich für welchen Einsatz besser eignet. Das Experiment zeigte, dass Apache Flink einen leichten Performance Vorteil in bestimmten Versuchen gegenüber Apache Spark gezeigt hat.

Jannik Bruhns

Title of the paper

Machine learning in the big data environment - experimental comparison of Apache Flink and Apache Spark

Keywords

Apache Flink, Apache Spark, Machine Learning, MLlib, Spark ML, Flink ML, Big Data, experiments, Support Vector Machine, MinMax Scaler, K-Means, Stochastic Outlier Selection, Multiple Linear Regression, Alternating Least Squares, implementation, comparison

Abstract

At the moment the big topic in computer science is "‘Artificial Intelligence"’, a subsection of this is "‘Machine Learning"’. The open source Big Data Frameworks Apache Flink and Apache Spark offer various libraries and interfaces such as machine learning. In this thesis the intent is to find out which framework is better for which assignment. The experiment showed that Apache Flink has a slighty performance advantage in certain attempts compared to Apache Spark.

Inhaltsverzeichnis

1 Einleitung **1**

2 Apache Spark **2**
2.1 Ökosystem 2
2.1.1 Bibliotheken und APIs 3
2.1.2 Cluster Managers 5
2.1.3 Resilient Distributed Dataset 7

3 Apache Flink **8**
3.1 Ökosystem 9
3.1.1 Bibliotheken und APIs 9
3.2 Dataflow Graphs 12
3.3 DataSet API (Batch Processing) 12
3.4 Bulk- und Delta-Iteration 13

4 Machine Learning **15**
4.1 Supervised Learning 15
4.1.1 Regression 15
4.1.2 Classification 16
4.2 Unsupervised Learning 16
4.2.1 Clustering 16
4.3 Pipelines 17
4.4 MLlib 20
4.4.1 Pipelines 21
4.5 FlinkML 22
4.5.1 Pipelines 23
4.6 ML-Algorithmen dieser Arbeit 24
4.6.1 Support Vector Machine (SVM) 24
4.6.2 MinMax Scaler 24
4.6.3 Alternating Least Squares (ALS) 26
4.6.4 K-Means 27
4.6.5 Multiple Linear Regression 27
4.6.6 Stochastic Outlier Selection 28

5 Experimente **29**
5.1 Aufbau der Experimente 29
5.1.1 Hypothesen 29

5.1.2 Cluster . . . 30
5.1.3 Algorithmen . . . 31
5.1.4 Daten . . . 31
5.2 Experimentelle Durchführung . . . 32
5.3 Auswertung der Experimente . . . 33
5.3.1 Support Vector Machine (SVM) . . . 33
5.3.2 MinMax Scaler . . . 35
5.3.3 Alternating Least Squares (ALS) . . . 38
5.3.4 K-Means . . . 40
5.3.5 Multiple Linear Regression (MLR) . . . 42
5.3.6 Stochastic Outlier Selection (SOS) . . . 45
5.3.7 Zusammenfassung und Bewertung der Ergebnisse . . . 48

6 Fazit **49**
6.1 Zusammenfassung . . . 49
6.2 Ausblick . . . 50

Literaturverzeichnis **55**

Abbildungsverzeichnis

2.1 Apache Spark Ecosystem https://databricks.com . . . 3
2.2 Spark Cluster Komponenten (SDC+16) . . . 6
2.3 Vereinfachter Spark Datenfluss (SDC+16) . . . 7

3.1 Apache Flink Ecosystem (Fli19b) . . . 9
3.2 Delta-Iteration (Fli19d) . . . 13

4.1 Pipeline Beispiel 1 (Spa18c) . . . 18
4.2 Pipeline Beispiel 2 (Spa18c) . . . 19

5.1 Support Vector Machine Laufzeit mit verschiedener Worker Anzahl . . . 33
5.2 Support Vector Machine Laufzeit mit verschieden großen Datensätzen . . . 34
5.3 MinMax Scaler Laufzeit mit verschiedener Worker Anzahl . . . 35
5.4 MinMax Scaler Laufzeit mit verschieden großen Datensätzen . . . 36
5.5 Alternating Least Squares Laufzeit mit verschiedener Worker Anzahl . . . 38
5.6 Alternating Least Squares Laufzeit mit verschieden großen Datensätzen . . . 39
5.7 K-Means Laufzeit mit verschiedener Worker Anzahl . . . 40
5.8 K-Means Laufzeit mit verschieden großen Datensätzen . . . 42
5.9 Multiple Linear Regression Laufzeit mit verschiedener Worker Anzahl . . . 43
5.10 Multiple Linear Regression Laufzeit mit verschieden großen Datensätzen . . . 44
5.11 Stochastic Outlier Selection Laufzeit mit verschiedener Worker Anzahl . . . 45
5.12 Stochastic Outlier Selection Laufzeit Anstieg bei verschieden großen Datensätzen 46
5.13 Stochastic Outlier Selection Laufzeit mit verschieden großen Datensätzen . . . 46

Tabellenverzeichnis

4.1 Apache Flink und Apache Spark Algorithmen Vergleich in Tabellenform . . . 25

5.1 Apache Flink und Apache Spark Konfiguration 32
5.2 Support Vector Machine Messergebnisse 35
5.3 MinMax Scaler Messergebnisse 37
5.4 Alternating Least Squares Messergebnisse 39
5.5 K-Means Versuch Ergebnisse von 8,16 und 32 Workern im direkten Vergleich. 41
5.6 K-Means Versuch Ergebnisse von 1,2 und 4 Workern im direkten Vergleich. . . 41
5.7 Multiple Linear Regression Messergebnisse 44
5.8 Stochastic Outlier Selection Messergebnisse 47

1 Einleitung

Für den Big Data Bereich gibt es verteilte Datenverarbeitung Frameworks wie z.B. Apache Spark oder Apache Flink. Sie funktionieren verteilt und arbeiten effizient bei endlichen Batchdaten und endlosen Datenströmen. Diese Daten können mit Maschine Learning Algorithmen analysiert werden um Zusammenhänge zu erschließen. Beim Streaming Dienst Netflix werden beispielsweise Algorithmen eingesetzt, die Benutzern Filme und Serien aufgrund ihres Sehverhaltens vorschlagen, die ihnen gefallen könnten. Die Batchverarbeitung ist der Kern von Apache Flink und Apache Spark. Es gibt zusätzlich noch Bibliotheken und APIs für den Einsatz von Machine Learning.
Das Ziel dieser Thesis ist es herauszufinden, ob Apache Spark oder Apache Flink sich für Machine Learning besser oder schlechter eignet. Eine Bewertung wird auf Grundlage von Experimenten abgegeben, verschiedene Machine Learning Algorithmen werden in unterschiedlichen Skalierungen mit verschieden großen Datensätzen ausgeführt. Bei den Experimenten wird die Laufzeit für die Verarbeitung von Algorithmen ermittelt.
In Kapitel 2 wird zunächst Apache Spark und in Kapitel 3 Apache Flink präsentiert. Im 4. Kapitel wird auf einige Teilbereiche von Machine Learning eingegangen. Danach werden beide Framework Machine Learning Bibliotheken - MLlib und FlinkML - vorgestellt. Des Weiteren werden die in den Experimenten verwendeten Algorithmen genauer erklärt. In Kapitel 5 werden Aufbau und Durchführung der Experimente beschrieben, die Ergebnisse werden aufgezeigt. Im 6. Kapitel wird die Thesis zusammengefasst und ein Ausblick auf weitere mögliche Experimente gegeben.

2 Apache Spark

Apache Spark ist ein Open-Source Framework für die parallele Verarbeitung von großen Datenmengen, besonders geeignet ist es für die iterative Verarbeitung von Algorithmen. Zu Beginn startete Apache Spark als Forschungsprojekt von AMP Laboratory der Universität von Kalifornien an Berkeley (Algorithms Machines and People Lab) und wurde durch seine schnelle Entwicklung in kurzer Zeit zu einem der Top-Open-Source-Projekte der Apache Software Foundation (vgl. GW17). Diese Geschwindigkeit entsteht durch Aufteilung der Ressourcen auf verschiedene Knoten, zudem wird durch Zwischenspeicherung von Daten, die für die Berechnung benötigt werden, im Speicher der einzelnen Cluster Knoten die Verarbeitung effizienter. Durch dieses In-Memory-Cluster-Computing kann Apache Spark iterative Algorithmen ausführen, da Programme Speicherpunkte überprüfen und darauf zurückgreifen können, ohne sie erneut von der Festplatte laden zu müssen. Zusätzlich unterstützt Apache Spark interaktive Abfragen und Streaming-Datenanalysen mit extrem hohen Geschwindigkeiten (vgl. BK16).

2.1 Ökosystem

Das Ökosystem von Apache Spark besteht aus verschiedenen teils unabhängigen Komponenten siehe Abbildung 2.1 zur Verarbeitung und Analyse von Daten und Datenströmen, dazu gehören unter anderem MLlib für Machine Learning und GraphX für die Graphenverarbeitung (vgl. FSW16).

2.1.1 Bibliotheken und APIs

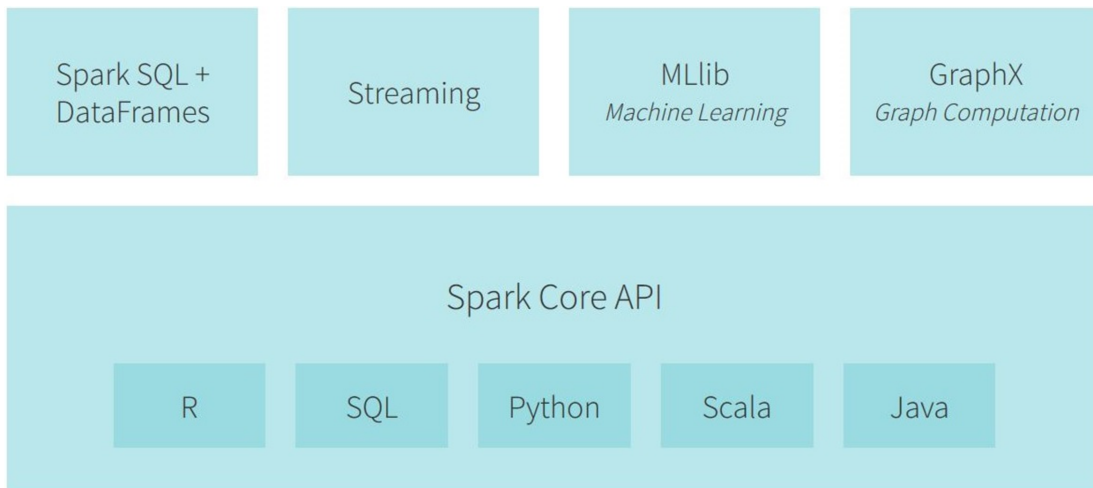

Abbildung 2.1: Apache Spark Ecosystem https://databricks.com

Auf die Komponenten von Abbildung 2.1 wird in den folgenden Abschnitten eingegangen:

- **Spark Core**
  Spark Core ist die Grundlage von Apache Spark. Es bietet eine einfache Programmierschnittstelle für die Verarbeitung großer Stückzahlen von Datensätze, die RDD-Schnittstelle. Spark Core wurde in Scala implementiert, bietet jedoch weitere Schnittstellen in Java, Scala, Python, SQL und R an. Diese Schnittstellen unterstützen viele Operationen u.a. Datentransformationen und Aktionen, die für Datenanalyse Algorithmen in höheren Bibliotheken essentiell sind. Darüber hinaus bietet Spark Core Hauptfunktionalität für In-Memory-Cluster-Computing Speicherverwaltung, Arbeitsplanung, Datenumstrukturierung und Fehlerbehebung. Verbesserungen im Spark Core führen zu entsprechenden Verbesserungen in den übergeordneten Bibliotheken, da diese Bibliotheken auf dem Spark Core basieren (vgl. SDC$^+$16).

- **Spark SQL + DataFrames**
  Spark SQL ist die Apache Spark Komponente für die strukturierte Datenverarbeitung. Sie bietet eine Programmierabstraktion namens DataFrames an und kann auch als verteilte SQL-Abfrage-Engine fungieren. Dies macht es möglich unveränderte Hadoop-Hive-Abfragen in vorhandenen Bereitstellungen und Daten bis zu 100-mal schneller ausführbar zu machen. Zusätzlich bietet Spark SQL eine leistungsstarke Integration mit dem restlichen Spark-Ökosystem (z. B. Integration der SQL-Abfrageverarbeitung in MLlib) (vgl. Spa19a).

- **Streaming**
  Streaming implementiert eine inkrementelle Datenstrom-Verarbeitung mit einem Modell namens „discretized streams“. Um Streaming über Apache Spark zu realisieren, werden die Eingabedaten in kleine Batches (z. B. alle 200 Millisekunden) geteilt, die regelmäßig mit einem Status RDDs gespeichert werden, um neue Ergebnisse zu erstellen. Die Ausführung von Streaming-Berechnungen über diesen Weg hat mehrere Vorteile gegenüber dem traditionellen verteilten Streaming-Systems. Durch die Verwendung der Datenherkunft ist die Fehlerbehebung kostengünstiger. Außerdem ist es dadurch möglich, Streaming mit Batches und iterativen Anfragen zu kombinieren (vgl. ZXW+16).

- **MLlib (Machine Learning Library)**
  Die *Machine Learning Library (MLlib)* wird durch allgemeine Lernalgorithmen und statistische Dienstprogramme gebildet. Zu den Hauptfunktionen zählen: Classification, Regression, Clustering, kollaboratives Filtern, Optimierung und Dimensionalitätsreduktion. Diese Bibliothek wurde speziell entwickelt, um die Machine Learning-Pipelines in Umgebungen von großer Dimension zu vereinfachen. In den letzten Versionen von Apache Spark wurde die MLlib Bibliothek in zwei Pakete aufgeteilt:
  MLlib, die auf RDDs aufgebaut ist und ML, die mit DataFrames arbeitet (vgl. GGRGGH17).
  Im Abschnitt 4.4 wird diese Bibliothek detaillierter betrachtet.

- **GraphX (Graph Computation)**
  GraphX ist eine Komponente von Apache Spark für die Graphen und Parallele-Graphen Berechnungen. Auf hohem Niveau erweitert GraphX die Spark-RDD, indem eine neue Graphen Abstraktion eingeführt wird: ein gerichteter Multigraph mit Eigenschaften an jedem Knoten und an jeder Kante. Zur Unterstützung der Graphenberechnung stellt GraphX eine Anzahl grundlegender Operatoren z. B. Subgraph, joinVertices und aggregateMessages, sowie eine optimierte Variante der Pregel-API bereit. Darüber hinaus

stellt GraphX eine stetig wachsende Sammlung von Graph Algorithmen und Vorlagen bereit, um graphenanalytische Aufgaben zu vereinfachen (vgl. Spa18b).

2.1.2 Cluster Managers

Apache Spark-Anwendungen laufen in unabhängigen Prozesssätzen in einem Cluster und werden von dem SparkContext Objekt im Hauptprogramm (Driver Program) koordiniert. Für die Ausführung in einem Cluster kann der SparkContext eine Verbindung zu verschiedenen Arten von Cluster-Managern herstellen, die Ressourcen anwendungsübergreifend zuweisen. Sobald die Verbindung hergestellt ist, erzeugt Apache Spark Executoren auf Worker Nodes(Knoten) im Cluster, welche als Prozesse die Berechnungen ausführen und Daten für Ihre Anwendung speichern. Der Anwendungscode wird zu den Executoren gesendet (definiert durch JAR- oder Python-Dateien, die an SparkContext übertragen werden). Danach kann SparkContext Aufgaben, die von den Executoren ausgeführt werden sollen, senden.
Zu dieser Architektur sind einige nützliche Dinge zu beachten:

- Jede Anwendung erhält eigene Executor-Prozesse, die für die gesamte Laufzeit der Anwendung aktiv bleiben und Aufgaben in mehreren Threads ausführen. Der Vorteil: Anwendungen werden voneinander isoliert, sowohl auf der Planungsseite (jeder Treiber plant seine eigenen Aufgaben) als auch auf der Executor-Seite (Aufgaben aus verschiedenen Anwendungen werden in verschiedenen Java Virtual Machines(JVMs) ausgeführt). Wiederum bedeutet dies jedoch auch, dass Daten nicht von verschiedenen Apache Spark-Anwendungen (Instanzen von SparkContext) gemeinsam genutzt werden können, ohne sie in ein externes Speichersystem zu speichern.

- Apache Spark ist vom Cluster-Manager abhängig. Solange Apache Spark Executor-Prozesse erwerben kann und diese miteinander kommunizieren, ist es relativ einfach, Apache Spark auch auf einem Cluster-Manager auszuführen, der auch andere Anwendungen unterstützt (z. B. Mesos / YARN).

- Das Driver Program muss während seiner gesamten Lebensdauer eingehende Verbindungen von seinen Executoren abhören und akzeptieren. Daher muss das Driver Program von den Worker Nodes über das Netzwerk adressierbar sein.

- Da der Driver Aufgaben im Cluster plant, sollte er in der Nähe der Worker Nodes ausgeführt werden, vorzugsweise im selben lokalen Netzwerk.

(vgl. Spa18a)

Apache Spark wurde so konzipiert, dass es effizient von einem bis zu mehreren Tausend Knoten-Berechnungen skalieren kann. Um dies zu erreichen und gleichzeitig maximal flexibel zu sein, kann Apache Spark auf verschiedenen Cluster Managern laufen (vgl. KKWZ15). Durch die YARN Kompatibilität ist Apache Spark auf vorhandenen Hadoop-Cluster ausführbar. Somit kann auf jede Hadoop-Datenquelle einschließlich HDFS, S3, HBase und Cassandra zugegriffen werden (vgl. BK16).

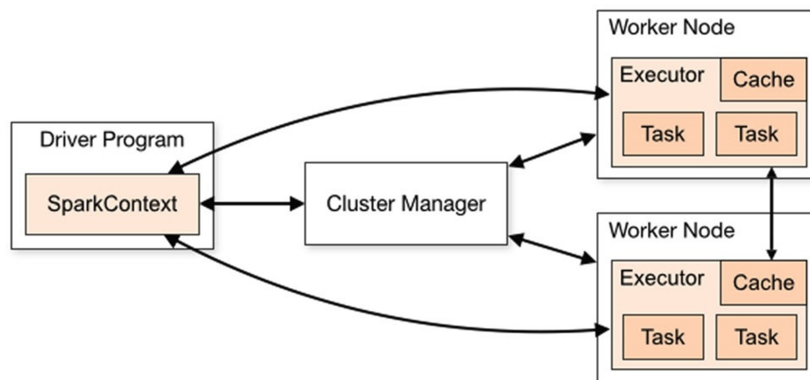

Abbildung 2.2: Spark Cluster Komponenten (SDC⁺16)

Das System unterstützt zurzeit drei Cluster Manager:

- Standalone – in Apache Spark integriert, für eine einfache Cluster Einrichtung,
- Apache Mesos – ein Manager für u.a. Hadoop MapReduce and Service Anwendungen,
- Hadoop YARN – der Ressourcen Manager in Hadoop 2,
- Kubernetes – ein Open-Source System für automatisierte Ausführung, Skalierung und Management für Containeranwendungen.

(vgl. Spa18a)

2.1.3 Resilient Distributed Dataset

Bei Resilient Distributed Datasets (RDDs) handelt es sich um eine schreibgeschützte Sammlung von Objekten, die auf verschiedene Maschinen aufgeteilt werden und bei einem Ausfall einer Partition neu erstellt werden können. Dies ermöglicht Apache Spark eine fehlertolerante Methode, um riesige Datenmengen in den Speicher zu laden. Neben dieser RDDs hat Apache Spark zusätzlich eine Serie von Operationen für RDDs definiert, die eine parallele Berechnung unterstützen. RDD Operationen können lose in zwei Kategorien unterteilt werden: Transformationen und Aktionen. Bei Transformationen wird ein RDD eines Typs A in ein RDD eines Typs B mit einer benutzerdefinierten Funktion umgewandelt. Beispiele für Transformationen sind map(), flatMap(), and filter(). Aktionen wiederum erfordern die Durchführung einer wirklichen Berechnung. Aktionen verarbeiten einen bestimmten RDD und führen zu einem bestimmten Typ von Ergebnis. Beispiele für Aktionen sind reduce() und collect(). Sowohl Transformationen als auch Aktionen werden parallel von Apache Spark ausgeführt. Abbildung 2.3 zeigt diesen Datenfluss in Apache Spark. Als Erstes werden Daten aus dem Dateisystem in einen RDD geladen. Nach dem Laden werden eine Reihe von Transformationen am RDD durchgeführt. Zum Schluss wird noch eine Aktion benutzt und das Programm beendet (vgl. SGA16).

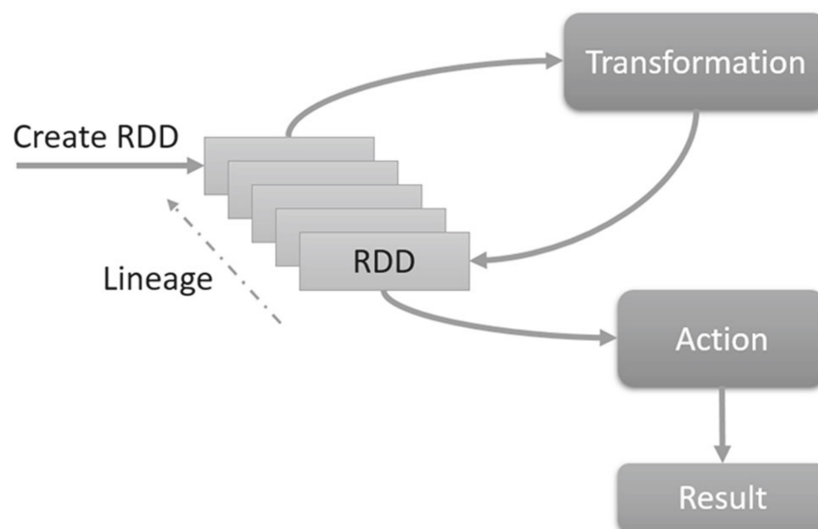

Abbildung 2.3: Vereinfachter Spark Datenfluss (SDC^{+}16)

3 Apache Flink

Apache Flink, früher als Forschungsprojekt mit dem Namen Stratosphere bekannt, ist ein Open-Source Framework, welches an drei Universitäten in Berlin gestartet ist und später ein Apache Top Level Projekt wurde. Der Kern bildet eine verteilte DataFlow-Engine, die es erlaubt sowohl Datenströme als auch Batchdaten zu verarbeiten. Die Batchverarbeitung konzentriert sich auf das Konzept eines DataSets - eine verteilte Sammlung mit den Elementen des zu verarbeiteten Datensatzes. Benutzer können spezifische funktionale Transformationen auf diesen DataSets anwenden wie z.B. map(), fiter(), reduce(). Programme in Flink werden „lazily" ausgeführt, das heißt, das Laden der Daten-Transformationen findet nicht sofort statt, sondern jede Operation wird erstellt und zum Programm Plan hinzugefügt. Operationen werden nur bei dem Aufruf der execute() Methode im Execution-Umgebungs-Objekt aufgerufen. Analog zur Abfrageoptimierung in Datenbanken wird das Programm in einen logischen Plan umgewandelt, kompiliert und durch einen kostenbasierten Optimierer bestmöglich gestaltet. Dieser wählt automatisch eine Ausführungsstrategie für das Programm anhand von verschiedenen Parametern wie z.B. Datengröße oder die Anzahl von Maschinen im Cluster. Der endgültige physische Plan ist somit eingeteilt und kann von der verteilten DataFlow-Engine ausgeführt werden, welche die Daten per Pipelining verarbeitet. Apache Flink erlaubt es dem Benutzer nicht, DataSets in den Arbeitsspeicher zwischenzuspeichern, bietet jedoch einen eigenen nativen Iterationsoperator zur Angabe von iterativen Algorithmen. Der Flink-Optimierer erkennt dies und fügt dem physischen Plan zwischengespeicherte Operationen zu, um sicherzustellen, dass keine Daten mit Schleifeninvarianten vom verteilten Dateisystem in jeder Iteration erneut gelesen werden. Im Gegensatz dazu implementiert Apache Spark Iterationen als reguläre For-Schleifen und führt sie durch Schleifenabwicklung aus (vgl. BSRM17).

3.1 Ökosystem

Das Ökosystem von Flink besteht aus mehreren Bibliotheken für allgemeine Anwendungsfälle der Datenverarbeitung. Die Bibliotheken sind normalerweise in eine API eingebettet und nicht vollständig in sich geschlossen. Daher können sie von allen Funktionen der API profitieren und in andere Bibliotheken integriert werden (vgl. Fli19j).

Der Core, bei Flink auch Runtime genannt, ist eine verteilte DataFlow-Engine, die DataFlow-programme ausführt. Ein Flink Runtime Programm ist ein gerichteter azyklischer Graph (DAG) von zustandsbehafteten Operatoren, die mit Datenströmen verbunden sind. In Flink gibt es zwei Haupt-APIs: Die DataSet-API für die Verarbeitung von endlichen Datensätzen, auch als Batchverarbeitung bezeichnet, sowie die DataStream-API für die Verarbeitung unbegrenzter Datenströme, oft als Stream-Verarbeitung bezeichnet. Sowohl die DataSet-API als auch die DataStream-API erstellen Laufzeitprogramme, die von der Engine ausgeführt werden können. Diese werden von den Bibliotheken erstellt, zu denen u.a. FlinkML für Machine Learning, Gelly für Graph-Verarbeitung und Table für SQL Operationen zählen (vgl. CEF^{+}17).

3.1.1 Bibliotheken und APIs

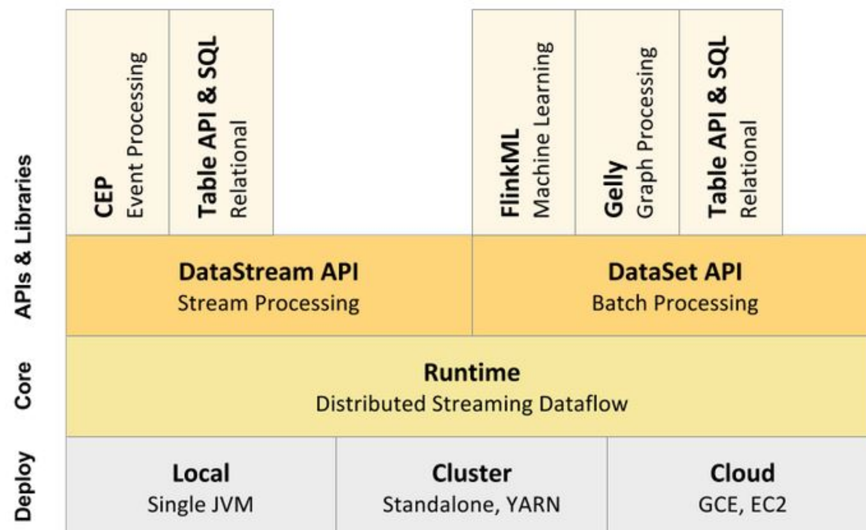

Abbildung 3.1: Apache Flink Ecosystem (Fli19b)

Auf die Komponenten von Abbildung 3.1 wird in den folgenden Abschnitten eingegangen:

- **Runtime**
  Die Runtime leitet eine physische Bereitstellung von Aufgaben und verwaltet deren kontinuierliche Ausführung. Wie bei den meisten verteilten Datenverarbeitungssystemen besteht Flink's Runtime aus einem JobManger, einem Masterprozess, der die Metadaten von aktiven Pipelines hält und koordiniert die Ausführung durch Kommunikation mit den Worker-Prozessen (TaskManager). Die Kommunikation zwischen Jobmanager und TaskManager folgt einem asynchrones RPC-basiertes Kommunikationsprotokoll, bestehend aus regelmäßigen Statusaktualisierungen (Heartbeats) für den JobManager und Zeitplanungsanfragen zurück zum TaskManager. Im Gegensatz zu Stapel Jobverwaltung, die Neukonfiguration und Koordination priorisiert, benutzt Flink eine einmalig langlaufende Zuordnung von Aufgaben. Das System ist jedoch flexibel für eine Neukonfiguration der Pipelines zu mehr oder weniger Worker-Prozessen und für eine Zuteilung des Anwendungsstatus auf Abruf. Mit diesem Ansatz wird der Verwaltungsaufwand minimiert und erlaubt es dennoch, weitere Anpassungen an Hardware und Software vorzunehmen oder teilweise auftretende Fehler zu beheben. Außerdem können durch die Pipeline Bereitstellung selbst Master-Ausfälle toleriert werden (vgl. CEF+17).

- **Complex Event Processing (CEP)**
  Die Mustererkennung ist ein sehr häufiger Anwendungsfall für die Ereignisstreamverarbeitung. Flink's CEP Bibliothek bietet eine API um Ereignismuster zu bestimmen, z.B. reguläre Ausdrücke oder Endliche Automaten. CEP ist in der DataStream-API von Flink integriert, sodass Muster auf DataStreams ausgewertet werden. Zu den Anwendungen von CEP gehören die Erkennung von Netzwerkeinbrüchen, die Überwachung von Geschäftsprozessen und die Erkennung von Betrug (vgl. Fli19j).

- **FlinkML**
  Die *FlinkML* Bibliothek stellt skalierbare MachineLearning-Algorithmen und eine intuitive Schnittstelle zu Verfügung. Sie enthält Algorithmen für Supervised Learning, Unsupervised Learning, Data Preprocessing, Recommendation und andere Dienstprogramme (vgl. GGRGGH17).
  Im Abschnitt 4.5 wird diese Bibliothek detaillierter betrachtet.

- **Gelly**
  Gelly ist die Graph API von Flink. Sie enthält eine Reihe von Methoden und Dienstprogrammen, die die Entwicklung von Graphenanalyse-Anwendungen in Flink vereinfachen sollen. In Gelly können Graphen transformiert und modifiziert werden, indem sie Funktionen auf hoher Ebene verwenden, die den Funktionen der Batchverarbeitungsschnittstelle ähneln. Gelly bietet Methoden zum Erstellen, Transformieren und Ändern von Graphen sowie eine Bibliothek mit Graphalgorithmen an(vgl. Fli19c).

- **Table API und SQL**
  Table API und SQL sind zwei relationale APIs von Flink für die einheitliche Stream- und Batchverarbeitung. Die Table-API ist eine in die Sprache integrierte Abfrage-API für Scala und Java, mit der Abfragen von relationalen Operatoren wie Auswahl, Filter und Join auf sehr intuitive Weise erstellt werden können. Gestellte Anfragen haben dieselbe Semantik und geben das gleiche Ergebnis aus, unabhängig davon, ob die Eingabe eine Batch-Eingabe (DataSet) oder eine Stream-Eingabe (DataStream) ist. Ohne Probleme kann zwischen den APIs und Bibliotheken gewechselt werden, die auf den Haupt APIs aufbauen. Somit kann z.B. ein Muster aus einem DataStream mithilfe der CEP-Bibliothek extrahiert werden und später darauf die Table-API angewendet werden, um die Muster zu analysieren. Ebenso kann eine Batchtabelle mithilfe einer SQL-Abfrage gescannt, gefiltert und aggregiert werden, bevor einen Gelly-Graph-Algorithmus auf diese vorverarbeiteten Daten ausgeführt wird (vgl. Fli19i).

3.2 Dataflow Graphs

Obwohl Benutzer Flink-Programme mit einer Vielzahl von APIs schreiben können, werden alle Flink-Programme schlussendlich in eine gemeinsame Darstellung (Dataflow Graph) kompiliert. Dieser wird von Flink's Runtime Engine ausgeführt. Ein Dataflow Graph ist ein gerichteter azyklischer Graph (DAG), der aus zustandsbehafteten Operatoren und Datenströmen besteht, diese repräsentieren die vom Operator erzeugten Daten, diese Daten können von Operatoren genutzt werden. Da DAGs datenparallel arbeiten, werden die Operatoren auf eine oder mehrere Instanzen parallelisiert, diese werden „subtasks" genannt. Streams werden wiederum in ein oder mehrere Stream Partitionen geteilt: eine Partition pro produzierender subtasks. Die zustandsbehafteten Operatoren, die als Sonderfall zustandslos sein können, implementieren die gesamte Verarbeitungslogik (z. B. Filter, Hash-Joins und Stream-Zeitfensterfunktionen). Viele dieser Operatoren sind Implementierungen von Lehrbuchversionen bekannter Algorithmen (vgl. CKE+15).

3.3 DataSet API (Batch Processing)

Ein eingeschränkter DataSet(Datensatz) ist ein Sonderfall eines unbegrenzten DataStreams. Ein Streaming-Programm, das alle Eingabedaten in einem Zeitfenster übermittelt, kann daraus ein Batch Programm erstellen. Die Batchverarbeitung sollte vollständig von Flink's Funktionen abgedeckt werden. Programme, die begrenzte Datensätze verarbeiten, haben Zugang zu zusätzlichen Optimierungen, effizienterer Buchhaltung für Fehlertoleranz und gestaffelte Zeitplanung. Flink geht die Batchverarbeitung wie folgt an: Batchverarbeitungen werden von derselben Runtime ausgeführt wie Streaming-Berechnungen. Die Runtime Ausführung sollte mit blockierenden DataStreams parametrisiert werden, um große Berechnungen in isolierte Stufen aufzulösen, die daraufhin nacheinander geplant werden. Die periodische Snapshotting Funktion ist deaktiviert, wenn der Anteil von Zusatzinformationen zu hoch ist. Stattdessen kann die Fehlerbehebung durch die Wiedergabe der verlorenen Stream Partition von dem letzten materialisierten Zwischenstrom (möglicherweise die Quelle) mögliche Fehler ausmerzen. Blockierungsoperatoren (z. B. Sortierungen) sind einfache Operatorimplementierungen, die zufällig blockiert werden, bis sie ihre gesamte Eingabe eingelesen haben. Die Runtime weiß nicht, ob ein Operator blockiert ist oder nicht. Diese Operatoren verwenden verwaltbare Speicher, die von Flink bereitgestellt werden (entweder auf oder außerhalb des JVM-Heapspeichers). Sie können bei Bedarf auf die Festplatte geschrieben werden, wenn die Eingabe die Speichergrenzen überschreitet. Eine gesonderte DataSet-API stellt bekannte Abstraktionen für Batchberechnungen bereit, nämlich eine begrenzte Fehlertoleranz-DataSet-Datenstruktur und

Transformationen in DataSets (z. B. Joins, Aggregationen, Iterationen). Eine Anfrageoptimierungsschicht wandelt ein DataSet-Programm in eine effiziente ausführbare Datei um (vgl. CKE+15).

3.4 Bulk- und Delta-Iteration

Flink bietet zwei Möglichkeiten, um iterative Teile eines DAGs auszuführen: Bulk-Iteration und Delta-Iteration. Bulk-Iteration berechnen immer wieder Zwischenergebnisse einer Iteration als Ganzes neu. In vielen Fällen jedoch konvergieren Teile des Zwischenzustands bei unterschiedlichen Geschwindigkeiten z.B. in der kürzesten Pfadberechnung mit einer Quelle in großen Graphen. In diesem Fall würde das System Ressourcen verschwenden, indem es immer wieder den Zwischenzustand neu berechnet einschließlich der Teile, die sich nicht mehr ändern (vgl. DXS+15).

Um dieses Problem zu beheben, gibt es den Spezialfall, der Delta-Iteration genannt wird. Delta-Iteration machen es sich zu Nutzen, dass bei der Berechnung nicht jedes Datenelement bei jedem Iterationsschritt aktualisiert wird. Sie benutzen Workset und Solution Set als Eingabe.

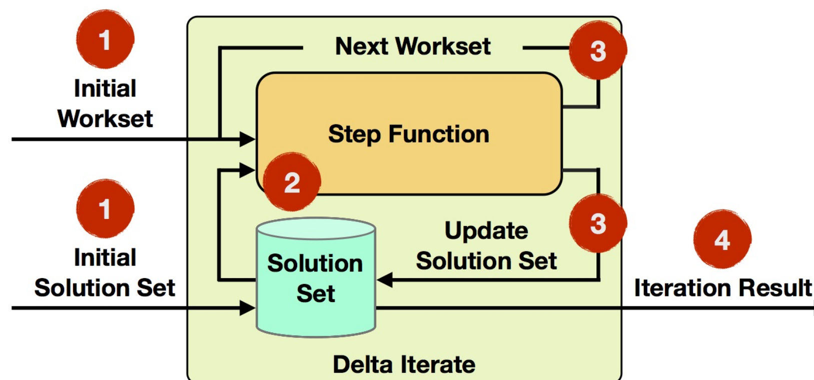

Abbildung 3.2: Delta-Iteration (Fli19d)

1. **Iteration Input:** Der erste Workset und Solution Set werden aus Datenquellen oder vorherigen Operatoren, als Eingabe für die erste Iteration, gelesen.
2. **Step-Funktion:** Die Step-Funktion wird bei jeder Iteration ausgeführt. Es ist ein beliebiger Dataflow, der aus Operatoren wie map, reduce, join, usw. besteht und von Ihrer spezifischen Aufgabe abhängig ist.
3. **Next Workset / Update Solution Set:** Der nächste Workset steuert die iterative Berechnung und wird in die nächste Iteration zurückgespeist. Außerdem wird der Solution Set aktualisiert und implizit weitergeleitet (er muss nicht neu erstellt werden). Beide Datensätze können von verschiedenen Operatoren der Schrittfunktion aktualisiert werden.
4. **Iterationsergebnis:** Nach der letzten Iteration wird der Solution Set als Eingabe für die folgenden Operatoren verwendet.

(vgl. Fli19d)

Ein schrumpfender Workset verbessert die Effizienz, indem konvergierte Subinstanzen vom Problem unberührt bleiben. Außerdem kann der Solution Set bei jedem Iterationsschritt geändert werden. Die Ausgabe der Delta-Iteration ist der Solution Set nach der letzten Iteration. Die Delta-Iteration endet entweder, wenn der Workset leer ist oder die maximale Anzahl der Iterationen erreicht wurde. Ein wichtiges Merkmal von Datenverarbeitungs-Frameworks ist, dass Aufgaben ohne weitere Änderungen am Programmcode verteilt werden. Die Aufgaben können lokal auf einem einzelnen Computer ausgeführt werden oder auf einem Cluster mit Hunderten oder Tausenden von Computern (vgl. VTK16).

4 Machine Learning

Machine Learning beschäftigt sich mit der Erkennung von Mustern in Daten und der Verwendung dieser gelernten Muster, um Vorhersagen bzw. Wissen aus dem Erlernten zu erstellen. Die meisten Machine Learning Algorithmen können in zwei Hauptkategorien eingeteilt werden: Supervised Learning und Unsupervised Learning.

4.1 Supervised Learning

Supervised Learning befasst sich mit dem Lernen einer Funktion (Mapping) aus einer Menge von Eingaben (Features) und Ausgaben. Der Lernvorgang vom Algorithmus erfolgt mit einem Trainingssatz von Ein- und Ausgabepaaren, der verwendet wird, um sich der Funktion anzunähern, dieses Training macht es dem Algorithmus möglich Assoziationen herzustellen. Ein typisches Anwendungsbeispiel ist die Handschrifterkennung. Weitere Teilbereiche sind Classification und Regression. Bei einer Classification wird versucht vorherzusagen, zu welcher Klasse ein Objekt gehört, ob z.B. ein Nutzer auf die Werbung klicken wird oder nicht. Die Regression versucht wiederum (reale) numerische Werte vorherzusagen, die häufig als abhängige Variablen bezeichnet werden, wie die Temperatur bei der Wettervorhersage (vgl. Fli19g).

Supervised Learning kann in mehrere Teilbereiche aufgeteilt werden. In dieser Thesis werden Regression und Classification weiter betrachtet.

4.1.1 Regression

Bei Regressionsproblemen wird versucht, Ergebnisse anhand von Eingaben aus einer kontinuierlichen Funktion vorherzusagen. Regression bedeutet, die Bewertung einer Variable basierend auf den Bewertungen einer anderen vorherzusagen Variable. Die vorherzusagende Variable heißt Kriteriumvariable und die Variable, von der die Vorhersage ausgeht, wird als Prädiktorvariable bezeichnet. Es besteht die Möglichkeit, dass es mehr als eine Prädikatorvarible gibt. In

diesem Fall wird die am besten passende Linie gefunden, die Regressionsgerade genannt wird (vgl. Des17).

4.1.2 Classification

Bei der Classification wird die Ausgabe in diskreten Ergebnissen prognostiziert. Classification als Teil von Supervised Learning benötigt Trainingsdaten. Dabei wird versucht, die Ergebnisse in Gruppen definierter Kategorien zu klassifizieren. Zum Beispiel werden anhand der angegebenen Merkmale die Aufzeichnungen von Personen in männlich oder weiblich klassifiziert. Ein weiteres Beispiel wäre die Vorhersage, die auf einem bestimmten Kundenverhalten basiert, z.B. ob der Kunde ein Produkt kaufen würde oder nicht (vgl. Des17).

4.2 Unsupervised Learning

Unsupervised Learning versucht gegenüber dem Supervised Learning Muster und Regelmäßigkeiten in Daten zu erkennen, ohne diese anhand von einem Trainingssatz zu erlernen. Hierfür wäre Clustering ein Beispiel, bei dem versucht wird Gruppierungen der Daten anhand der beschreibenden Funktionen zu ermitteln. Eine weitere Möglichkeit zur Verwendung wäre die Feature Selection, bei der die Hauptkomponenten analysiert werden (vgl. Fli19g).

4.2.1 Clustering

Die Begrifflichkeit des Clustering kann am Besten durch ein Beispiel verständlich gemacht werden. Äpfel, Bananen, Zitronen und Kirschen in einem Obstkorb, sollen in Gruppen eingeteilt werden. Bei der Betrachtung der Farben der Früchte können diese in zwei Gruppen eingeteilt werden: Äpfel und Kirschen (rote Farbgruppe), Bananen und Zitronen (gelbe Farbgruppe). Anschließend können die Früchte basierend auf ihrer Größe noch weiter differenziert werden. Anders als beim Supervised Learning sind beim Clustering keine Trainingsdaten und keine zu prognostizierende Variable vorhanden. Vielmehr besteht die Aufgabe darin, mehr über die Eigenschaften zu erfahren und die Datensätze basierend auf Eigenschaften zu gruppieren (vgl. Des17).

4.3 Pipelines

Pipelines im ML-Kontext können als Operationsketten betrachtet werden, die einige Daten als Eingabe haben, eine Anzahl von Veränderungen an diesen Daten durchführen und dann die veränderten Daten wieder ausgeben. Dies ist entweder als Eingabe einer predictor Funktion eines Lernmodells oder als normale Ausgabe veränderter Daten für eine andere Aufgabe möglich. Vergleichbar ist diese Funktion mit dem Pipes und Filter Architekturmuster. ML-Pipelines sind oft komplizierte Operationen und können zu Fehlerquellen für End-zu-End Learning Systemen werden.
Der Zweck einer ML-Pipeline besteht darin ein Framework zu schaffen, das die eingeführte Komplexität der Operationskette verwaltet. Pipelines sollten für Entwickler eine Vereinfachung darstellen, verkettete Veränderungen zu definieren. Diese Veränderungen werden auf die Trainingsdaten angewendet, um am Ende eine Funktion zu erhalten, die zum Training eines Lernmodells benutzt werden. Außerdem können dann dieselben verketteten Veränderungen an unbekannten Testdaten durchgeführt werden. Zusätzlich sollten Pipelines die Cross-Validation und Modellauswahl für diese Operationsketten vereinfachen (vgl. Fli19e).

Ein Transformer ist eine Abstraktion, die Feature-Transformer und erlernte Modelle enthält. Technisch gesehen implementiert ein Transformer die Methode transform(), die einen DataFrame in ein anderen DataFrame konvertiert. Im Allgemeinen geschieht dies durch Anhängen einer oder mehrerer Spalten. Zum Beispiel könnte ein Feature-Transformer einen DataFrame übernehmen, eine Spalte lesen (z. B. Text), in eine neue Spalte abbilden (z. B. Feature-Vektoren) und einen neuen DataFrame mit angehängter zugeordneter Spalte ausgeben. Ein Lernmodell kann einen DataFrame verwenden, die Spalte mit Merkmalsvektoren lesen, die Bezeichnung für jeden Merkmalsvektor vorhersagen und einen neuen DataFrame mit vorhergesagten Kennzeichnungen ausgeben, die als Spalte angehängt werden (vgl. Spa18c).

Ein Estimator abstrahiert das Konzept eines Lernalgorithmus oder eines Algorithmus, der Daten anpasst oder mit diesen trainiert. Dieser implementiert eine Methode fit(), die einen DataFrame akzeptiert und ein Model erzeugt, das wiederum ein Transformer ist. Beispielsweise ist ein Lernalgorithmus wie LogisticRegression ein Estimator, die Methode fit() trainiert ein LogisticRegressionModel und dieses Model ist somit ein Transformer (vgl. Spa18c).

Funktionsweise

Eine Pipeline ist eine Sequenz von Stufen und jede Stufe ist entweder ein Tranformer oder ein Estimator. Der Reihe nach werden diese Stufen ausgeführt und der Eingabe-DataSet wird beim Durchlaufen jeder einzelnen Stufe umgewandelt. Bei Tranformer-Stufen wird die transform() Methode auf dem DataSet aufgerufen. Für Estimator-Stufen wird die fit() Methode aufgerufen, um einen Transformer, der Teil des PipelineModels oder der angepassten Pipeline wird, zu erstellen. Danach wird die transform() Methode von diesem Transformer auf dem DataSet aufgerufen.
Die folgende Abbildung veranschaulicht die Trainingszeit einer Pipeline, die nachgehend durch ein einfaches Workflow Textdokument beschrieben wird (vgl. Spa18c).

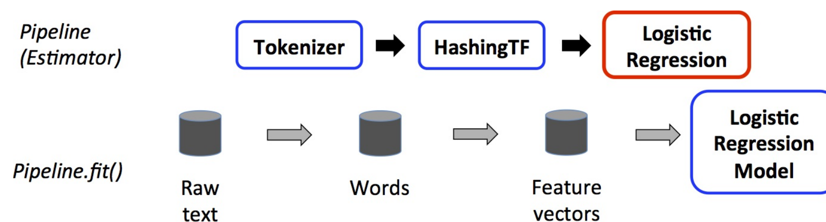

Abbildung 4.1: Pipeline Beispiel 1 (Spa18c)

In Abbildung 4.1 präsentiert die obere Reihe eine Pipeline mit drei Stufen. Die ersten beiden (Tokenizer und HashingTF) Stufen sind Transformer (blau) und die dritte (LogisticRegression) Stufe ist ein Estimator (rot). Die untere Reihe stellt Daten dar, die durch die Pipeline fließen, wobei die Darstellung der Zylinder die DataSets verkörpern. Die fit() Methode der Pipeline wird auf dem ursprünglichen DataSet aufgerufen, der Rohtextdokumente und Beschriftungen enthält. Mit Hilfe der transform() Methode des Tokenizer wird das Rohtextdokument in Wörter eingeteilt und fügt diese dann als neue Spalte an den DataSet. Bei der HashingTF.transform() Methode wird die Spalte „Wörter“ in Feature-Vektoren konvertiert und auch als neue Spalte an den DataSet hinzugefügt. LogisticRegression ist ein Estimator, das bedeutet es wird zunächst von der Pipeline LogisticRegression fit() aufgerufen, um ein LogisticRegressionModel zu erzeugen. Wenn die Pipeline mehr Estimators enthält, würde sie die transform() Methode von LogisticRegressionModel auf dem DataSet aufrufen, bevor der DataSet an die nächste Stufe übergeben wird.
Da eine Pipeline ein Estimator ist, wird von ihr die fit() Methode aufgerufen, die ein PipelineModel (Transformer) erstellt. Dieses PipelineModel wird zur Testzeit verwendet. Auf Abbildung 4.2 wird diese Verwendung veranschaulicht (vgl. Spa18c).

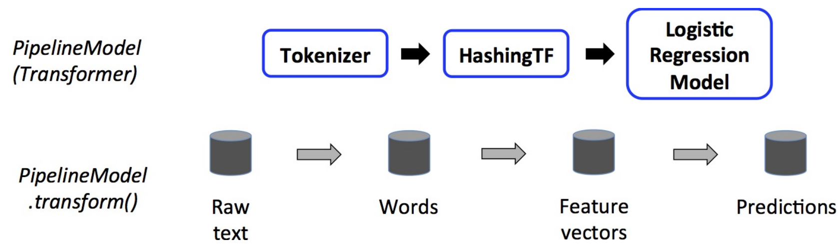

Abbildung 4.2: Pipeline Beispiel 2 (Spa18c)

In Abbildung 4.2 hat das PipelineModel die gleiche Anzahl von Stufen wie die ursprüngliche Pipeline, nur sind alle Estimators der ursprünglichen Pipeline hier Transformers. Wenn die transform() Methode vom PipelineModel auf ein Test-Dataset aufgerufen wird, werden die Daten der Reihe nach durch die angepasste Pipeline geleitet. Die transform() Methode jeder Stufe aktualisiert das Dataset und übergibt es an die nächste Stufe. Pipelines und PipelineModels helfen sicherzustellen, dass Trainings- und Test-Daten identische Verarbeitungsschritte durchlaufen (vgl. Spa18c).

4.4 MLlib

MLlib ist Apache Spark's Machine Learning Bibliothek, deren Ziel es ist, Machine Learning skalierbar und einfach benutzbar zu machen. Sie bietet flexible und skalierbare Implementierungen von einer Vielzahl von Machine Learning Komponenten, wie z.B. Ensemble Learning und Principal Component Analysis (PCA) über Optimierung und Clusteranalyse. Darüber hinaus können auch verteilte Verarbeitungen durch parallele Verarbeitung und Unterstützung von Big Data-Tools verteilte Architekturen, genutzt werden. Diese Kriterien verringern die Verarbeitungszeit und gleichzeitig wird Zeit geschaffen oder eingespart, um Analyseergebnisse zu interpretieren. Sehr wichtig wird dies, wenn viele Vorhersagen berechnet werden. Die verteilte Architektur kann von manchen Big Data-Tools zum Vorteil genutzt werden, indem Machine Learning Komponenten voneinander getrennt werden, um die Gesamtlaufzeit zu verkürzen. MLlib profitiert von der Einbindung in mehrere im Apache Spark-Ökosystem verfügbare Softwarekomponenten, wie Spark GraphX, Spark SQL und Spark Streaming. Hinzu kommt noch eine Vielzahl gut organisierter Dokumentationen, einschließlich Codebeispielen, die öffentlich und frei für die Machine Learning Community zur Verfügung stehen (vgl. ABLT17).

Apache Spark teilt seine ML-Bibliothek in zwei Pakete auf:

- spark.mllib enthält die ursprüngliche API, die auf RDDs aufgebaut ist.
- spark.ml bietet eine übergeordnete API, die auf DataFrames aufgebaut ist, um ML-Pipelines zu erstellen.

Die primäre Machine Learning-API für Apache Spark ist die DataFrame-basierte API im spark.ml-Paket, da ab Apache Spark 2.0 die RDD-basierten APIs im Paket spark.mllib in den Wartungsmodus gewechselt haben. Die APIs im spark.mllib Paket werden dann nur noch solange existieren, bis alle Funktionen ins spark.ml-Paket implementiert wurden und daraufhin komplett entfernt werden.
Apache Spark hat sich dafür entschieden, weil DataFrames eine benutzerfreundlichere API als RDDs bieten. Zu den vielen Vorteilen von DataFrames gehören Spark-Datenquellen, SQL / DataFrame-Abfragen, Tungsten- und Catalyst-Optimierungen sowie einheitliche APIs für alle Sprachen. DataFrames ermöglichen praktische ML-Pipelines und insbesondere Feature-Transformationen (vgl. Spa18d).

4.4.1 Pipelines

In diesem Abschnitt werden die abweichenden Konzepte der Pipeline-API gegenüber der Apache Flink Pipeline-API beschrieben

DataFrame
Ein DataFrame ist ein DataSet welches aus benannten Spalten besteht. Es ist gleichbedeutend mit einer Tabelle in einer relationalen Datenbank, jedoch mit umfangreicheren Optimierungen ausgestattet. DataFrames können aus einer Vielzahl von Quellen erstellt werden, z. B. aus strukturierten Datendateien, Tabellen in Hive, externen Datenbanken oder vorhandenen RDDs. Die DataFrame-API ist in Scala, Java, Python und R verfügbar. In Scala und Java wird ein DataFrame durch einen Datensatz von Zeilen dargestellt. In der Scala-API ist DataFrame einfach ein Typalias von Dataset [Row]. In Java-API müssen Benutzer jedoch Dataset <Row> verwenden, um einen DataFrame darzustellen. Ein DataFrame kann entweder implizit oder explizit aus einer regulären RDD erstellt werden (vgl. Spa19b).

DAG Pipelines
Die Stufen einer Pipeline sind als geordnetes Array spezifiziert. Die hier erklärten Pipelines sind alle lineare, d. H. jede Stufe benutzt die Daten, die von der vorherigen Stufe erzeugten wurden. Außerdem ist es möglich nicht lineare Pipelines zu erstellen, solange der DataFlow-Graph ein Directed Acyclic Graph (DAG) ist. Dieser Graph wird derzeit implizit basierend auf den Namen der Eingabe- und Ausgabespalten jeder Stufe (meistens Parameter) angegeben. Bildet die Pipeline eine DAG, müssen die Stufen in topologischer Reihenfolge angegeben werden.

Laufzeitprüfung
Pipelines können keine Typprüfung zur Kompilierungszeit verwenden, da sie mit DataFrames unterschiedlicher Typen arbeiten. Pipelines und PipelineModels führen stattdessen eine Laufzeitprüfung durch, bevor die Pipeline tatsächlich ausgeführt wird. Für diese Typprüfung wird das DataFrame Schemata benutzt, welches eine Beschreibung der Datentypen jeder Spalte im DataFrame enthält.

(vgl. Spa18c)

4.5 FlinkML

Die Machine Learning (ML) Bibliothek für Flink(*FlinkML*)ist entwickelt worden, um skalierbare ML-Tools in Flink zu integrieren. Das Ziel ist es ein System zu entwerfen und zu implementieren, das skalierbar ist und somit Probleme unabhängig von der Datenmenge verarbeiten kann. Eine wichtige Bedeutung für Entwickler von ML-Systemen ist die Menge an Glue-Code, die sie gezwungenermaßen bei der Implementierung von End-zu-End-ML-Systemen schreiben müssen. Aus diesem Grund soll FlinkML den Entwicklern helfen den Glue-Code auf ein Minimum zu reduzieren. Das Flink-Ökosystem bietet die passende Umgebung bei der Bewältigung dieses Problems, indem seine skalierbaren ETL-Funktionen einfach im selben Programm kombiniert werden können. Die robuste Pipeline Entwicklung macht es möglich, keine andere Technologie für die Datenaufnahme und die Manipulation von Daten zu benötigen. FlinkML macht es sind selbst zum Ziel eine Bibliothek zu entwickeln, die besonders benutzerfreundlich ist. Dafür sorgen ausführliche Dokumentationen mit Beispielen für jeden Teil des Systems. Dies soll Entwicklern ermöglichen mit bekannten Programmierungs-Konzepten und Terminologien schnell und einfach ihre eigene ML Pipeline schreiben zu können. Im Gegensatz zu anderen Datenverarbeitungssystemen nutzt Flink In-Memory Data Streaming um iterative Verarbeitungsalgorithmen, die in ML üblich sind, auszuführen. FlinkML ermöglicht es Datenwissenschaftlern, ihre Modelle lokal zu testen und Teilmengen von Daten zu verwenden. Anschließend wird dann derselbe Code genutzt, um ihre Algorithmen in einer größeren Cluster Umgebung auszuführen. FlinkML ist von Scikit-Learn und MLib von Apache Spark inspiriert worden und besitzt daher eine ähnliche API Struktur (vgl. Vas16).

4.5.1 Pipelines

In diesem Abschnitt werden die abweichenden Konzepte der Pipeline-API gegenüber der Apache Spark Pipeline-API beschrieben.

Typen und Typensicherheit Neben den Fit- und Transform-Operationen bietet der StandardScaler auch eine Fit- und Tranform-Operation für Eingaben vom Typ LabeledVector. Dies ermöglicht es, den Algorithmus mit oder ohne beschrifteter Eingaben zu benutzen. Dies passiert automatisch und wird je nach Typ der Eingabe, die der Fit- und Transform-Operationen übergebenden wird entschieden. Die korrekte implizite Operation wird vom Compiler in Abhängigkeit vom Eingabetyp ausgewählt. Wenn die fit() oder transform() Methode mit Typen ausgeführt wird die nicht unterstützt werden, kommt es vor dem Starten des Auftrags zu einem Laufzeitfehler. Für diese Art von Fehlern wurde bewusst die Laufzeit-Exception dem Compiler-Fehler vorgezogen, um dem Entwickler bessere Informationen über den Fehler zu geben (vgl. Fli19e).

Verkettung Die Verkettung wird durch den Aufruf von chainTransformer() oder chainPredictor() für ein Objekt einer Klasse erreicht, die Transformer implementiert. Diese Methoden geben ein ChainedTransformer- bzw. ein ChainedPredictor-Objekt zurück. Wie bereits erwähnt können ChainedTransformer-Objekte weiter verkettet werden, während ChainedPredictor-Objekte dies nicht können. Diese Klassen sorgen für die Übertragung der Fit-, Transform- und Predict-Operation für die aufeinanderfolgenden Transformer oder eines Transformers und eines Predictors. Sie wirken auch rekursiv, wenn die Länge der Kette größer als zwei ist, da jeder ChainedTransformer eine Fit- und Transform-Operation definiert, die mit weiteren Transformern oder einem Predictor weiter verkettet werden kann. Es ist wichtig zu wissen, dass sich Entwickler und Benutzer bei der Implementierung ihrer Algorithmen keine Gedanken über die Verkettung machen müssen. All dies wird automatisch von FlinkML übernommen (vgl. Fli19e).

4.6 ML-Algorithmen dieser Arbeit

Die nächsten drei beschriebenen Algorithmen sind in beiden Machine Learning Bibiotheken (Apache Spark und Apache Flink) vorhanden und ein Algorithmus pro Kategorie wurde für die Durchführung des Experimentes gewählt.

4.6.1 Support Vector Machine (SVM)

Support Vector Machine kann sowohl für Regressions- als auch für Klassifizierungsaufgaben verwendet werden. Die häufigste Art der Verwendung sind jedoch die Klassifizierungsziele. Der SVM Algorithmus hat die Aufgabe, eine Hyperebene in einem N-dimensionalen Raum (N - die Anzahl von Merkmalen) zu finden, die die Datenpunkte eindeutig klassifiziert. Es gibt viele mögliche Hyperebenen, die ausgewählt werden können, um die beiden Klassen von Datenpunkten voneinander zu trennen. Das Ziel ist es eine Ebene zu finden, die den maximalen Rand aufweist, das heißt den maximalen Abstand zwischen Datenpunkten beider Klassen. Durch die Maximierung des Randabstands wird eine gewisse Verstärkung erzielt, sodass zukünftige Datenpunkte sicherer klassifiziert werden können.

4.6.2 MinMax Scaler

MinMax Scaler gehört zum Data Preprocessing, die Aufgabe dieses Scaler besteht darin ein DataSet zu skalieren, sodass alle Werte zwischen einem vom Benutzer angegebenen Bereich [min, max] liegen. Falls der Benutzer keinen bestimmten Minimal- und Maximalwert für den Skalierungsbereich angibt werden die Merkmale des Eingabedatensatzes so verändert, dass sie im Intervall [0,1] liegen.
Gegeben eine Menge von Eingangsdaten $x_1, x_2, \ldots x_n$
mit minimalem Wert:$x_{min} = min(x_1, x_2, ..., x_n)$
and maximum value:$x_{max} = max(x_1, x_2, ..., x_n)$
Der skalierte DataSet $z_1, z_2, \ldots, z_n$ wird zu: $z_i = \frac{x_i - x_{min}}{x_{max} - x_{min}}(max - min) + min.$
Dabei sind min und max die vom Benutzer angegebenen minimalen und maximalen Werte des zu skalierenden Bereichs (vgl. Fli19f).

Die hier gezeigte Tabelle vergleicht und gibt einen Überblick der mitgelieferten Algorithmen der beiden ML-Bibliotheken.

Apache Flink	Apache Spark
Classification	
Support Vector Machine k-Nearest Neighbors Join	Support Vector Machine Logistic Regression Decision tree classifier Random forest classifier Gradient-boosted tree classifier Multilayer perceptron classifier One-vs-Rest classifier Naive Bayes
Recommendation	
Alternating Least Squares	Alternating Least Squares
Regression	
Multiple Linear Regression	Linear Regression Generalized linear regression Decision tree regression Random forest regression Gradient-boosted tree regression Survival regression Isotonic regression
Preprocessing	
MinMaxScaler PolynomialFeatures StandardScaler Splitter	MinMaxScaler PolynomialExpansion StandardScaler TF-IDF Word2Vec CountVectorizer FeatureHasher Discrete Cosine Transform StringIndexer IndexToString OneHotEncoder OneHotEncoderEstimator ...
Clustering	
K-means	K-means Latent Dirichlet allocation Bisecting k-means Gaussian Mixture Model
Outlier selection	
StochasticOutlierSelection	-

Tabelle 4.1: Apache Flink und Apache Spark Algorithmen Vergleich in Tabellenform

4.6.3 Alternating Least Squares (ALS)

Der Alternating Least Squares (ALS) ist ein Recommendation Algorithmus, der eine gegebene Matrix R in zwei Faktoren U und V faktorisiert $R \approx U^T V$. Die unbekannte Zeilendimension wird als Parameter für den Algorithmus angegeben und als latente Faktoren bezeichnet. Die Matrizen U und V können als Benutzer- bzw. Artikelmatrix bezeichnet werden, da die Matrixfaktorisierung im Rahmen der Recommendation verwendet wird. Die i-te Spalte der Benutzermatrix wird mit ui bezeichnet und die i-te Spalte der Artikelmatrix ist vi. Die Matrix R kann als Bewertungsmatrix mit $(R)_{i,j} = r_{i,j}$ bezeichnet werden.
Um die Benutzer- und Artikelmatrix zu finden, wurde das folgende Problem gelöst:
mit λ als Regulierungsfaktor, wobei nui die Anzahl der vom Benutzer bewerteter Elemente und nvj, wie oft der Artikel j bewertet wurde. Dieses Regularisierungsschema zur Vermeidung von overfitting wird Weighted-λ-Regularization genannt.

$$\arg\min_{U,V} \sum_{\{i,j|r_{i,j} \neq 0\}} \left(r_{i,j} - u_i^T v_j\right)^2 + \lambda \left(\sum_i n_{u_i} \|u_i\|^2 + \sum_j n_{v_j} \|v_j\|^2\right)$$

Durch fixieren einer der Matrizen U oder V erhält man eine quadratische Form, die direkt gelöst werden kann. Die Lösung des modifizierten Problems wird garantiert die Gesamtkostenfunktion monoton verringern. Indem dieser Schritt abwechselnd auf die Matrizen U und V angewendet wird, kann die Matrixfaktorisierung iterativ verbessert werden.
Die Matrix R wird in spärlichen Darstellungen als Tupel von (i, j, r) angegeben, wobei i den Zeilenindex, j den Spaltenindex und r den Matrixwert an Position (i, j) bezeichnet (vgl. Fli19a).

4.6.4 K-Means

K-Means-Clustering gehört zum Typ Unsupervised Learning, der verwendet wird, wenn nicht gekennzeichnete Daten vorhanden sind (Daten ohne definierte Kategorien oder Gruppen). Das Ziel dieses Algorithmus ist es Gruppen in den Daten zu finden, wobei die Anzahl der Gruppen durch die Variable K dargestellt wird. Der Algorithmus arbeitet iterativ, um jeden Datenpunkt basierend auf den bereitgestellten Merkmalen einer von K Gruppen zuzuweisen. Datenpunkte werden, basierend auf der Ähnlichkeit der Features, gruppiert. Die Ergebnisse des K-Means-Clustering-Algorithmus:

- Die errechneten Schwerpunkte eines K-Cluster, mit denen neue Daten beschriftet werden können. Zum Beispiel bei der Beschriftung von Trainingsdaten (jeder Datenpunkt ist einem einzelnen Cluster zugeordnet).
- Anstatt Gruppen zu definieren, bevor die Daten angesehen werden, ermöglicht die Clusterung das Finden und Analysieren von Gruppen.

Jeder Schwerpunkt eines Clusters ist eine Sammlung von Merkmalswerten, die die resultierenden Gruppen definieren. Durch die Untersuchung der Gewichtungspunkte des Zentroid-Merkmals kann qualitativ interpretiert werden, welche Art von Gruppe jeder Cluster darstellt. Die letzten beiden Algorithmen sind in Apache Spark nicht mitgeliefert, daher wurden Implementationen von Drittanbietern verwendet, um diese trotzdem mit den mitgelieferten Algorithmen von Apache Flink vergleichen zu können.

4.6.5 Multiple Linear Regression

Multiple Linear Regression ist eine statistische Technik, die mehrere interpretative Variablen verwendet, um das Ergebnis einer Antwortvariable vorherzusagen. Das Ziel der Multiple Linear Regression besteht darin, die Beziehung zwischen den interpretativen Variablen und den Antwortvariablen zu modellieren.
Eine einfache lineare Regression ist eine Funktion, die es einem Analytiker oder Statistiker ermöglicht, auf der Grundlage der Informationen, die über eine andere Variable bekannt sind, Vorhersagen über eine Variable zu treffen. Die lineare Regression kann nur verwendet werden, wenn zwei kontinuierliche Variablen vorhanden sind - eine unabhängige und eine abhängige Variable. Die unabhängige Variable ist der Parameter, mit dem die abhängige Variable oder das Ergebnis berechnet wird. Um eine Beziehung zu verstehen, in der mehr als zwei Variablen vorhanden sind, wird eine Multiple linear Regression verwendet. Diese wird verwendet, um eine mathematische Beziehung zwischen einer Anzahl von Zufallsvariablen zu bestimmen. Mit

anderen Worten, MLR untersucht, wie mehrere unabhängige Variablen zu einer abhängigen Variablen gehören. Sobald jeder der unabhängigen Faktoren zur Vorhersage der abhängigen Variablen bestimmt wurde, können die Informationen von mehreren Variablen verwendet werden, um eine genaue Vorhersage über die Auswirkung auf die Ergebnisvariable zu erstellen. Das Modell erstellt eine Beziehung in Form einer geraden Linie (linear), die alle einzelnen Datenpunkte am besten approximiert.

4.6.6 Stochastic Outlier Selection

Ein Outlier ist eine oder mehrere Beobachtungen, die quantitativ von der Mehrheit des Datensatzes abweichen und Gegenstand weiterer Untersuchungen sein können. Stochastic Outlier Selection ist ein Unsupervised Outlier Selection Algorithmus, der eine Menge von Vektoren als Eingabe verwendet. Der Algorithmus wendet eine affinitätsbasierte outlier-selection an und gibt für jeden Datenpunkt eine Outlierwahrscheinlichkeit aus. Intuitiv wird ein Datenpunkt als Outlier betrachtet, wenn die anderen Datenpunkte eine unzureichende Affinität aufweisen. Die Erkennung von Outliern findet in einer Reihe von Bereichen Anwendung, z. B. Protokollanalyse, Betrugserkennung, Geräuschbeseitigung, Neuheitserkennung, Qualitätskontrolle und Sensorüberwachung (vgl. Fli19h).

5 Experimente

In diesem Kapitel werden die Experimente ausgeführt, um die Performance der Machine Learning Bibliotheken von Apache Flink und Apache Spark zu ermitteln und zu vergleichen.

5.1 Aufbau der Experimente

An Hand der Experimente sollen die Stärken und Schwächen beider Frameworks verdeutlicht werden. Dazu werden Hypothesen aufgestellt und überprüft bei welchen Algorithmen und bei welcher Skalierung diese zutreffen. Die hierbei verwendeten Cluster und Daten werden in den weiteren Abschnitten beschrieben.

5.1.1 Hypothesen

Die Experimente bestehen aus 5 Hypothesen, diese werden zur Feststellung der Performance herangezogen.

1. Apache Spark und Apache Flink brauchen für die Verarbeitung einer x-mal so großen Datenmenge x-mal so lang.
2. Apache Spark und Apache Flink brauchen mit x-facher Worker Anzahl eine 1 durch x-fache Durchführungsdauer.
3. Apache Flink ist bei größeren Datenmengen schneller als Apache Spark.
4. Apache Spark teilt Daten besser auf mehrere Worker auf als Apache Flink.
5. Apache Flink arbeitet mit weniger Worker schneller im Vergleich zu Apache Spark.

Begründung für die Wahl der Hypothesen

Die erste Hypothese wurde in erster Linie aus logischer Sicht gewählt: wird die Anzahl der Eingabedaten verdoppelt, verdoppelt sich die Dauer der Verarbeitung. Diesen Punkt greift auch die zweite Hypothese auf: wird die Anzahl der verwendeten Worker bei gleicher Eingabe verdoppelt, wird sich die Verarbeitungszeit wahrscheinlich halbieren.
Apache Flink wurde als Streaming Framework entwickelt, bei der in der Regel viel größere Datenmengen als bei der Batchverarbeitung anfallen. Auf dieser Grundlage kann davon ausgegangen werden, dass Apache Flink bei großen Datenmengen einen Vorteil gegenüber Apache Spark hat; dies wurde in Hypothese drei festgehalten.
Apache Spark benutzt immer die volle verfügbare Rechenleistung, bei Apache Flink muss dies vorher konfiguriert werden, auf dieser Aussage basiert Hypothese vier.
Die Annahme ist, dass Apache Flink bei mehr Workern langsamer als Apache Spark arbeitet. Daraus könnte resultieren, dass Flink beim Einsatz von einer kleinen Anzahl von Workern effektiver arbeitet.

5.1.2 Cluster

Die Experimente wurden auf Amazon EC2 t2.micro Clustern ausgeführt, die jeweils einen vCPU mit 2.5 GHz der Intel Xeon Family besitzen und mit 1 GB Arbeitsspeicher ausgerüstet sind. Diese Cluster laufen mit Ubuntu Server 18.04 LTS (64-Bit) Betriebssystem auf einer 8 GB SSD. Für die Ausführung der Experimente wurden die zu dieser Zeit Stabilen Releases von Spark 2.4.1 und Flink 1.7.2 verwendet, die direkt auf den Clustern liefen. Beide Frameworks benutzen Java 8 von Oracle und die Algorithmen wurden in Scala Version 2.11 geschrieben. Bei den Experimenten wurde bei beiden Frameworks ein Master- und 1-32 Slave-Worker benutzt. Des weiteren wird in den folgenden Abschnitten der Begriff „Worker“ als Synonym von Sparks Worker-Knoten und Flinks Taskmanager genannt. Die Java Virtual Machine (JVM) Heap Size wurde auf 500MB begrenzt, um OutOfMemoryError vorzubeugen.

Begründung für die Wahl der Cluster

Um bessere Ergebnisse zu erzielen wurden möglichst viele Computer in einem Cluster benutzt; dies wurde mit dem Amazon EC2 Cluster erreicht. Jedoch konnte nur eine sehr geringe Rechenleistung eingesetzt werden, um keine hohen Kosten zu generieren. Amazon Web Services bietet 750 Stunden pro Monat für ein Jahr als kostenloses Kontingent an, welches in einem Monat für die Ausführung der Experimente im ganzen Umfang genutzt wurde.

5.1.3 Algorithmen

Für die Experimente wurden folgende Machine Learning Algorithmen benutzt:

- Support Vector Machine aus dem Bereich Classification
- MinMax Scaler aus dem Bereich Data Preprocessing
- Alternating Least Squares aus dem Bereich Recommendation
- K-Means aus dem Bereich Clustering
- Multiple Linear Regression aus dem Bereich Regression
- Stochastic Outlier Selection aus dem Bereich Outlier Selection

Eine detaillierte Beschreibung der verwendeten Algorithmen ist im Kapitel 4.6 dieser Arbeit zu finden.

Begründung für die Wahl der Algorithmen

Es wurden typische Machine Learning Algorithmen verwendet, die aus den Teilbereichen: Classification, Data Preprocessing, Recommendation, Clustering, Regression und Outlier Selection stammen. Die in beiden Frameworks vorhandenen Algorithmen aus diesen Bereichen wurden ausgewählt, fehlende integrierte Implementationen wurden von Drittanbietern herangezogen.

5.1.4 Daten

Für die Verarbeitung der Algorithmen wurden verschiedenste Daten genutzt. Diese stammen aus realen Bereichen, die im folgenden Abschnitt ausgeführt werden. Der Alternating Least Squares Algorithmus wurde mit Daten des MovieLens Datasets ausgeführt; dieser beinhaltet 20 Millionen Bewertungen von 138.000 Benutzern, die 27.000 Filme bewertet haben (vgl. Gro16). Das UCI Machine Learning Repository stellt 473 Datasets für alle Bereiche des Machine Learning zur Verfügung. Aus diesem wurde für die Ausführung des Kmeans Algorithmus im Bereich Clustering ein Dataset mit 2 Millionen Zeilen an Messdaten des Stromverbrauches eines Haushaltes verwendet. Diese wurden über einen Zeitraum von fast 4 Jahren alle 60 Sekunden aufgezeichnet (vgl. Rep10). Für die restlichen vier Algorithmen wurde ein weiteres Dataset aus dem UCI Machine Learning Repository verwendet. Dieser beinhaltet 0,58 Millionen Zeilen an Daten aus kartographischen Aufzeichnungen der Wälder aus dem Roosevelt National Forest in Nord Colorado.

Begründung für die Wahl der Daten

Für die Wahl der Daten wurde in erste Linie auf die Größe des Datasets geachtet, um eine höhere Laufzeit zu garantieren und um bei höherer Skalierung größere Unterschiede feststellen zu können. Eine hohe Anzahl von Attributen spielte zusätzlich eine Rolle, um die Datasets für mehrere Algorithmen benutzen zu können. Es hätten ohne Probleme größere Datasets ausgewählt werden können, jedoch war dies mit dem begrenzt eingesetzten Arbeitsspeicher nicht möglich. Zuletzt wurde noch auf das Dateiformat und die Formatierung der Daten geachtet, um keine unnötige Zeit mit Konvertierung zu verbrauchen.

5.2 Experimentelle Durchführung

Das Experiment besteht aus der Ausführung der sechs Algorithmen. An Hand der Ergebnisse jedes einzelnen Algorithmus werden Hypothesen gewählt, die diese widerlegen oder bestätigen. Bei allen nachfolgenden Diagrammen sind Werte von 0 bzw. nicht angezeigte Punkte als abgebrochener Versuch zu werten, bei dem es zu keiner erfolgreichen Ausführung des Algorithmus kam. Apache Spark und Apache Flink wurden mit den folgenden Parametern ausgeführt:

Apache Flink		Apache Spark	
jobmanager.heap.size	500m	spark.executor.memory	500m
taskmanager.heap.size	500m		

Tabelle 5.1: Apache Flink und Apache Spark Konfiguration

Alle Algorithmen wurden mit Apache Spark und Apache Flink jeweils mit 1,2,4,8,16 und 32 Worker und die Anzahl der Worker jeweils mit 100% danach mit 50%, zuletzt mit 25% der Daten durchgeführt. Für beide Frameworks wurde der Standalone Modus verwendet. Die verwendeten Algorithmen wurden mit denselben (voreingestellten) Parametern ausgeführt, um diese bestmöglich miteinander zu vergleichen. Jede Messung wurde dreimal durchgeführt um Fehlermessungen zu vermeiden.

5.3 Auswertung der Experimente

5.3.1 Support Vector Machine (SVM)

Der Algorithmus Support Vector Machine verwendet den Datensatz mit 580.000 Zeilen an Daten. Da SVM mit Trainings- und Test-Daten arbeitet, wurde der Datensatz, wie es üblich ist, aufgeteilt, in diesem Fall im Verhältnis 60/40. Die in der Grafik gezeigte Anzahl der Daten entspricht somit den Trainingsdaten. In der folgenden Grafik wurden die aufgezeichneten Messdaten in einem Diagramm zur Veranschaulichung dargestellt.

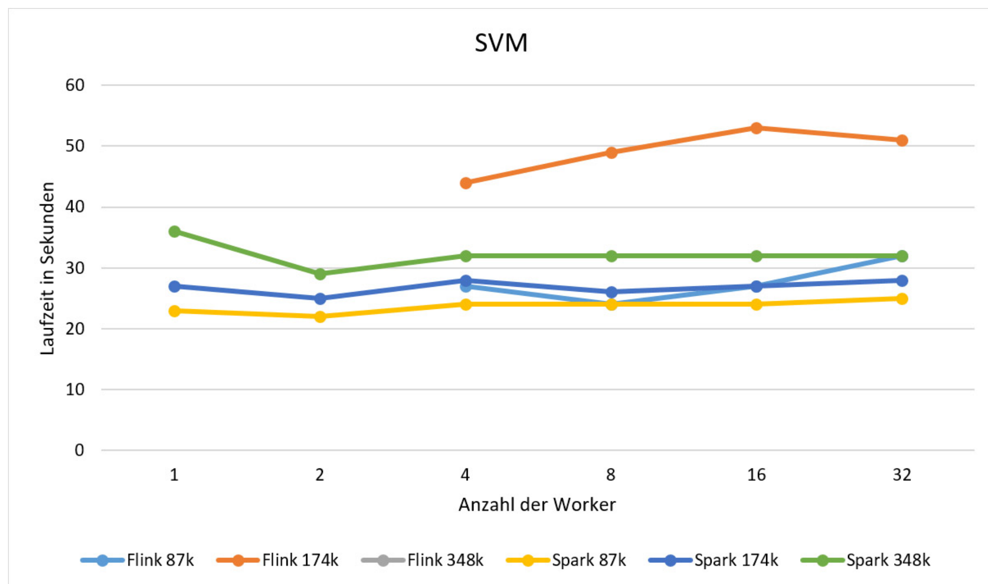

Abbildung 5.1: Support Vector Machine Laufzeit mit verschiedener Worker Anzahl

In der Abbildung 5.1 wurde der SVM Algorithmus in sechs verschiedenen Skalierungen mit 3 verschieden großen Daten ausgeführt. An der Y-Achse ist die Gesamtlaufzeit in Sekunden angegeben, an der X-Achse die Anzahl der verwendeten Worker. Die farbigen Linien repräsentieren verschieden große Datensätze mit dem jeweils Apache Spark und Apache Flink den Algorithmus ausgeführt hat. Die Ausführung der Daten mit der Größe 348k mit dem Framework Flink kam mit keiner Anzahl an Workern zu einem erfolgreichem Abschluss; es musste bei allen Versuchen nach ca. 30 Minuten manuell abgebrochen werden. Dies kam außerdem noch bei den Ausführungen mit einem und zwei Workern bei den weiteren beiden Datensätzen der Größe 174k und 87k vor. Somit konnte Flink bei dem Einsatz von einem oder zwei Workern diesen Algorithmus nicht ausführen. Bei der Analyse der Abbildung 5.1 liegen

die Laufzeiten zwischen 20 und ca. 50 Sekunden, was die Hypothese *„Apache Spark und Apache Flink brauchen für die Verarbeitung einer x-mal so großen Datenmenge x-mal so lang.“* und die Hypothese *„Apache Spark und Apache Flink brauchen mit x-facher Worker Anzahl eine 1 durch x-fache Durchführungsdauer.“* widerlegt, da kein linearer Anstieg sichtbar ist. Zu Hypothese *„Apache Flink arbeitet mit weniger Worker schneller im Vergleich zu Apache Spark“* ist klar zu sagen, dass es in diesem Experiment, bei Flink unter Verwendung von wenigen Workern, zu keinem Ergebnis kam und somit die Hypothese widerlegt ist. Eine Bestätigung der Hypothese *„Apache Spark teilt Daten besser auf mehrere Worker auf als Apache Flink.“* ist deutlich beim Vergleich des 174k Datensatzes zu sehen: hier ist Apache Spark fast doppelt so schnell im Vergleich zu Apache Flink. Die einzige Linie die sich von den anderen abhebt, ist die von Flink mit der Datensatz Größe von 174k. Das spricht deutlich gegen die Hypothese *„Apache Flink ist bei größeren Datenmengen schneller als Apache Spark.“* spricht. Apache Spark ist bei der Größe 87k nur 9% schneller, hingegen bei der Größe 174k knapp 76% schneller als Apache Flink. Zusätzlich ist zu sehen, dass bei dem größten Datensatz gar kein Ergebnis entstand. Dies ist noch besser: auf Abbildung 5.2 zu sehen. Hier wurden die schnellsten Zeiten pro Datensatz nebeneinander gestellt. Wie in der Grafik zuvor an der Y-Achse die Laufzeit des Algorithmus, nur hier auf der X-Achse die drei gewählten Datensätze.

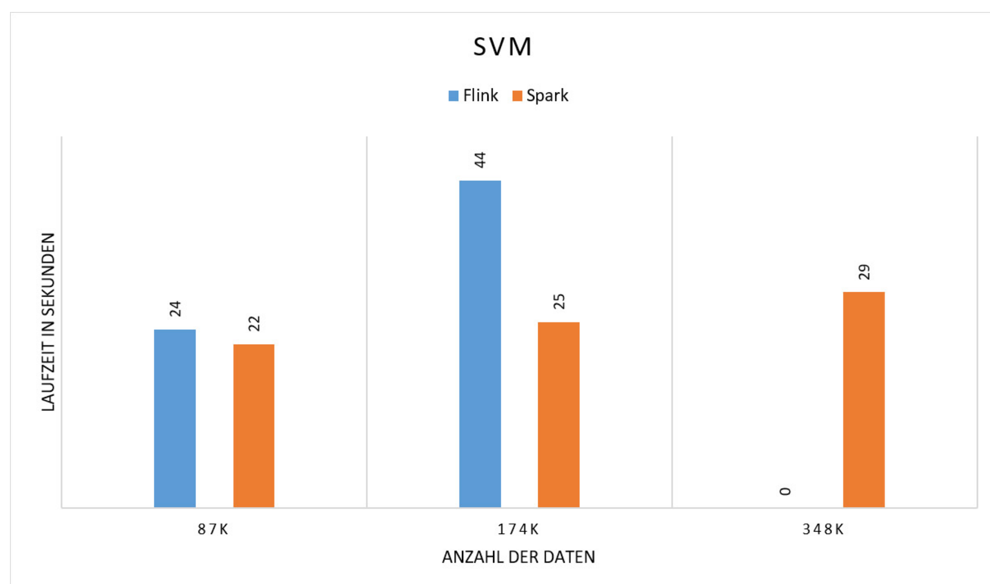

Abbildung 5.2: Support Vector Machine Laufzeit mit verschieden großen Datensätzen

		Worker Anzahl					
	Datensatz	1	2	4	8	16	32
Apache Flink	348k	-	-	-	-	-	-
Apache Spark	348k	36	29	32	32	32	32
Apache Flink	174k	-	-	44	49	53	51
Apache Spark	174k	27	25	28	26	27	28
Apache Flink	87k	-	-	27	24	27	32
Apache Spark	87k	23	22	24	24	24	25

Tabelle 5.2: Support Vector Machine Messergebnisse

5.3.2 MinMax Scaler

Das Experiment mit dem MinMax Scaler wurde mit dem vollen Datensatz in der Größe 580.000 Zeilen an Daten ausgeführt. Es folgt eine Grafik in Form eines Diagramms, in der die Messdaten eingeordnet werden.

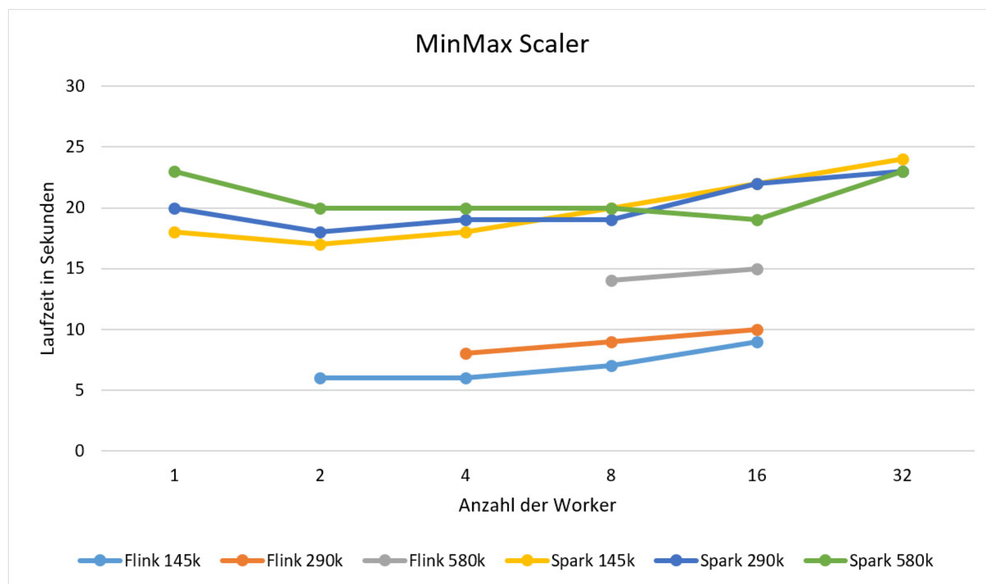

Abbildung 5.3: MinMax Scaler Laufzeit mit verschiedener Worker Anzahl

Wie auch in Abbildung 5.3 zu sehen, ist der MinMax Scaler Algorithmus in sechs verschiedenen Skalierungen mit jeweils drei verschieden großen Datensätzen ausgeführt worden. Auch hier zeigt die X-Achse die Laufzeit des jeweiligen Algorithmus in Sekunden an und die

Y-Achse die Anzahl der genutzten Worker. Die farbigen Linien bilden die Frameworks mit der entsprechenden Größe ab. Wie im ersten Versuch konnte Apache Flink einige Verarbeitungen nicht beenden, dies kam bei allen drei Datensatzgrößen beim Einsatz von einem und 32 Workern vor, außerdem noch bei den beiden größeren Datensätzen mit 2 Workern und beim größten verwendeten Datensatz mit der Ausführung von 4 Workern. Bei der Auswertung der Messdaten fiel sofort auf, dass Apache Flink, wenn es zu einem erfolgreichem Abschluss kam, immer schneller als Apache Spark war. Bei allen drei Apache Spark Ausführungen liegen die Messdaten sehr nah beieinander, woraus kein lineares Wachstum folgt. Hingegen lässt sich bei Apache Flink bei den Größen 290k und 540k ein Ansatz von vordoppelter Laufzeit erkennen. Für die Hypothese *„Apache Spark und Apache Flink brauchen für die Verarbeitung einer x-mal so großen Datenmenge x-mal so lang.“* gilt somit, dass sie nicht zutrifft. Die Hypothese *„Apache Spark und Apache Flink brauchen mit x-facher Worker Anzahl eine 1 durch x-fache Durchführungsdauer.“* lässt sich ebenso leicht widerlegen, da alle sechs Linien fast waagerecht verlaufen und nicht eine Steigung von zwei widerspiegeln. Bei der Verwendung von allen Worker Skalierungen zeigte Apache Flink eine schnellere Laufzeit als Apache Spark. Dies verifiziert die Hypothese *„Apache Flink arbeitet mit weniger Worker schneller im Vergleich zu Apache Spark.“* und wiederlegt zugleich die Hypothese *„Apache Spark teilt Daten besser auf mehrere Worker auf als Apache Flink.“*.

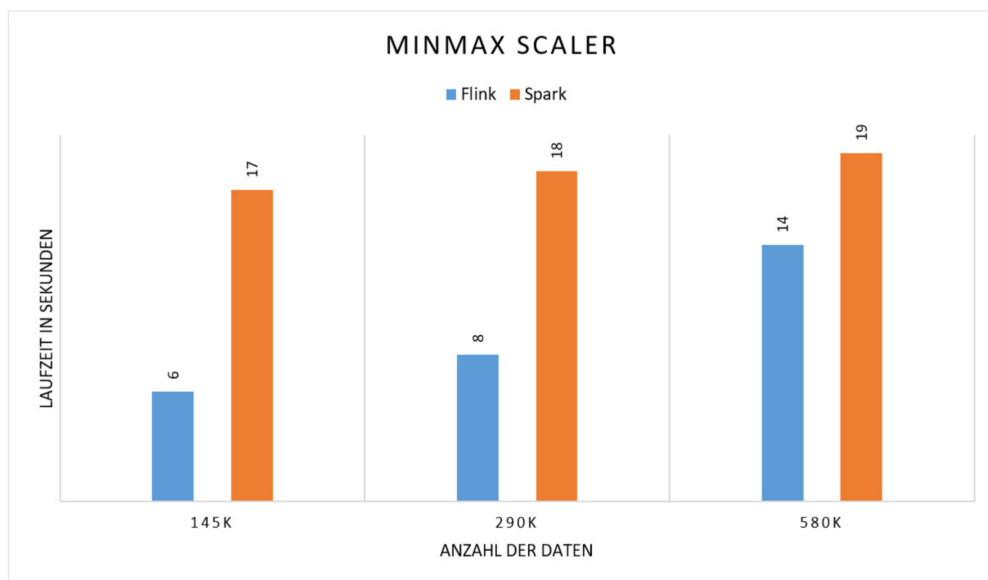

Abbildung 5.4: MinMax Scaler Laufzeit mit verschieden großen Datensätzen

Für den direkten Performance Vergleich wurde Abbildung 5.4 herangezogen. Hier wurden die besten Laufzeiten gegenübergestellt. Apache Flink schafft es bei dem Datensatz der Größe 580k 36% schneller als Apache Spark zu sein. Diese Laufzeitverbesserung kann sogar noch gesteigert werden: bei der Größe 290k konnte eine 125% schnelle Durchführungsdauer gemessen werden und bei der Größe 145k sogar eine Steigerung von 183%. Schlussendlich zeigt Apache Flink bei kleiner, mittlerer und großen Datenmenge bessere Ergebnisse als Apache Spark. Dies bestätigt Hypothese *„Apache Flink ist bei größeren Datenmengen schneller als Apache Spark.“*.

		Worker Anzahl					
	Datensatz	1	2	4	8	16	32
Apache Flink	580k	-	-	-	14	15	-
Apache Spark	580k	23	20	20	20	19	23
Apache Flink	290k	-	-	8	9	10	-
Apache Spark	290k	20	18	19	19	22	23
Apache Flink	145k	-	6	6	7	9	-
Apache Spark	145k	18	17	18	20	22	24

Tabelle 5.3: MinMax Scaler Messergebnisse

5.3.3 Alternating Least Squares (ALS)

Der Alternating Least Squares Algorithmus wurde mit 20 Millionen Bewertungen ausgeführt. In diesem Versuch konnte Apache Flink und Apache Spark alle Durchläufe beenden. Es folgt ein Liniendiagramm in dem die Messdaten veranschaulicht werden.

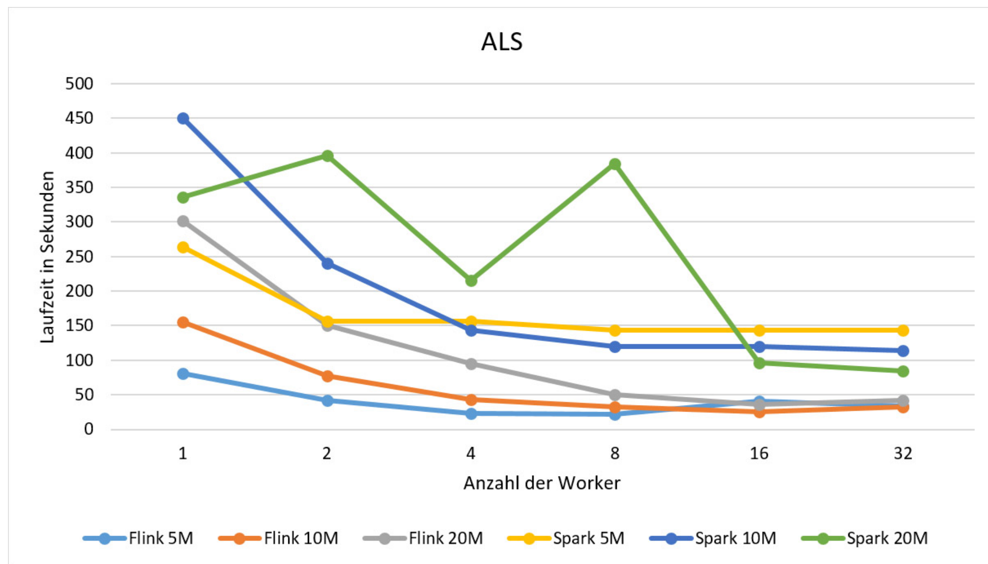

Abbildung 5.5: Alternating Least Squares Laufzeit mit verschiedener Worker Anzahl

Wie in Abbildung 5.5 zu sehen ist, konnte Apache Flink in allen Versuchen Apache Spark in puncto Performance übertreffen. Apache Flinks Laufzeiten verlaufen bei der Verwendung von 1,2 und 4 Workern annähernd linear. Bei Apache Spark ist nur das Muster zu erkennen, dass der größere Datensatz im Vergleich zu den anderen beiden von Apache Spark am schnellsten abgeschlossen wurde und der kleine bzw. mittlere Datensatz einen linearen Ansatz andeutet, dann jedoch in die waagerechte Position wechselt. Die Hypothese *„Apache Spark und Apache Flink brauchen für die Verarbeitung einer x-mal so großen Datenmenge x-mal so lang."* ist widerlegt und für die oben genannten Bereiche bestätigt. Die Halbierung der Laufzeit bei doppelter Verwendung von Workern ist bei allen drei Größen von Datensätze von Apache Flink beim Schritt von einem zu zwei Workern festgestellt worden. Somit ergibt sich eine Teilbestätigung der Hypothese *„Apache Spark und Apache Flink brauchen mit x-facher Worker Anzahl eine 1 durch x-fache Durchführungsdauer."*. Da bei allen Messungen Apache Flink schneller als Apache Spark performt ist Hypothese *„Apache Spark teilt Daten besser auf mehrere Worker auf als Apache Flink."* widerlegt und Hypothese *„Apache Flink arbeitet mit weniger Worker schneller im Vergleich zu Apache Spark."* bestätigt.

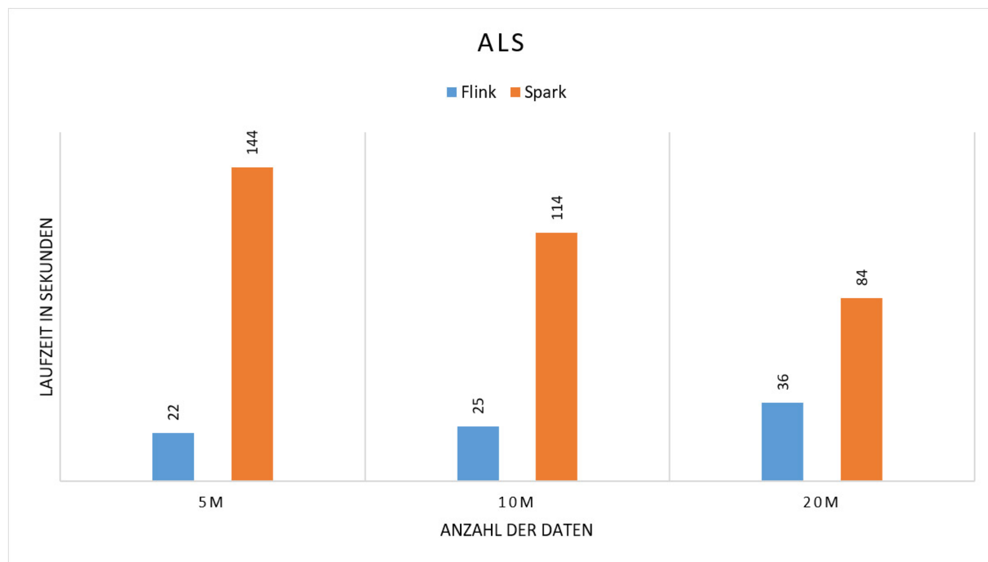

Abbildung 5.6: Alternating Least Squares Laufzeit mit verschieden großen Datensätzen

In Abbildung 5.6 ist die schnellste Laufzeit je Framework für die drei Datensätze gegenüber gestellt. Hypothese *„Apache Flink ist bei größeren Datenmengen schneller als Apache Spark.“* ist bestätigt, weil Apache Flink bei allen Messungen schneller war. Beim größten Datensatz war Apache Flink nur 133% schneller gegenüber Apache Spark, hingegen war beim mittleren Apache Flink 356% schneller und beim kleinsten verwendeten Datensatz sogar 554% schneller.

		Worker Anzahl					
	Datensatz	1	2	4	8	16	32
Apache Flink	20M	302	151	95	50	36	42
Apache Spark	20M	336	396	216	384	96	84
Apache Flink	10M	155	77	43	33	25	33
Apache Spark	10M	450	240	144	120	120	114
Apache Flink	5M	81	42	23	22	41	34
Apache Spark	5M	264	156	156	144	144	144

Tabelle 5.4: Alternating Least Squares Messergebnisse

5.3.4 K-Means

Das Clustering des K-Means Algorithmus wurde mit 2 Millionen Zeilen an Daten durchgeführt. Es folgt ein Diagramm mit den aufgezeichneten Messdaten.

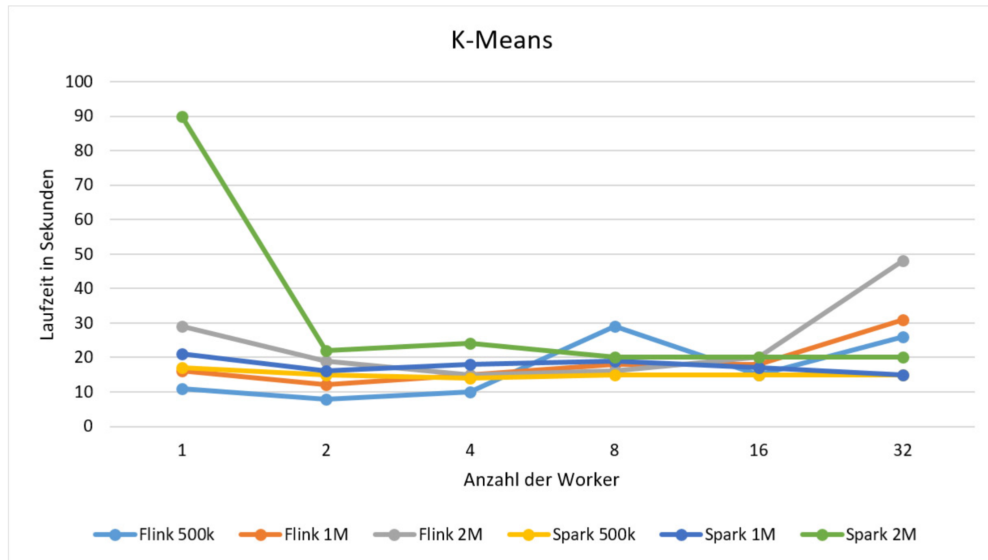

Abbildung 5.7: K-Means Laufzeit mit verschiedener Worker Anzahl

Die in Abbildung 5.7 dargestellten Ergebnisse zeigen eine waagerechte Tendenz mit einigen Ausreißern. Daraus kann geschlussfolgert werden, dass sich die Laufzeit nicht linear zur Datenmenge verhält. Somit ist Hypothese *„Apache Spark und Apache Flink brauchen für die Verarbeitung einer x-mal so großen Datenmenge x-mal so lang.“* nicht verifiziert. Zusätzlich kann keine Abhängigkeit von Laufzeit zu Workeranzahl festgestellt werden; dies widerlegt auch Hypothese *„Apache Spark und Apache Flink brauchen mit x-facher Worker Anzahl eine 1 durch x-fache Durchführungsdauer.“*.

		Worker Anzahl		
	Datensatz	8	16	32
Apache Flink	2M	16s	20s	48s
Apache Spark	2M	20s	20s	20s
Apache Flink	1M	18s	18s	31s
Apache Spark	1M	19s	17s	15s
Apache Flink	500k	29s	15s	26s
Apache Spark	500k	15s	15s	15s

Tabelle 5.5: K-Means Versuch Ergebnisse von 8,16 und 32 Workern im direkten Vergleich.

Wie in 5.5 zu sehen sind sechs von neun Ausführungen von Apache Spark schneller als die von Apache Flink, somit ist Hypothese *„Apache Spark teilt Daten besser auf mehrere Worker auf als Apache Flink.“* bestätigt.

		Worker Anzahl		
	Datensatz	1	2	4
Apache Flink	2M	29s	19s	15s
Apache Spark	2M	90s	22s	24s
Apache Flink	1M	16s	12s	15s
Apache Spark	1M	21s	16s	18s
Apache Flink	500k	11s	8s	10s
Apache Spark	500k	17s	15s	14s

Tabelle 5.6: K-Means Versuch Ergebnisse von 1,2 und 4 Workern im direkten Vergleich.

In neun von neun Fällen ist Apache Flink mit dem Einsatz von einem, zwei und vier Workern schneller, siehe 5.6. Das bestätigt die Hypothese *„Apache Flink arbeitet mit weniger Worker schneller im Vergleich zu Apache Spark.“*.

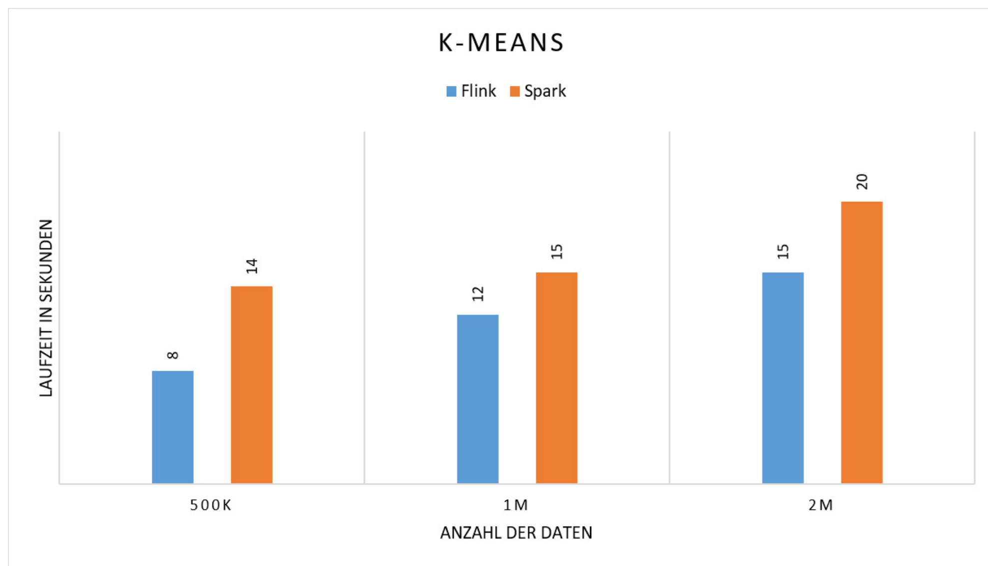

Abbildung 5.8: K-Means Laufzeit mit verschieden großen Datensätzen

Hypothese *„Apache Flink ist bei größeren Datenmengen schneller als Apache Spark.“* wird durch 5.8 bestätigt. Hier ist Apache Flink mit dem kleinsten getesteten Datensatz 75% schneller, bei dem mittleren noch 25% und bei dem größten Datensatz 33%.

5.3.5 Multiple Linear Regression (MLR)

Multiple Linear Regression ist ein Supervised Learning Algorithmus, der mit Trainings- und Test-Daten arbeitet; üblicherweise wird der Datensatz aufgeteilt. In diesem Fall wurde das Verhältnis 70/30 der Daten mit der Größe 580k gewählt. Ein weiteres Mal konnte Apache Flink bei der Verwendung von einem Worker keinen der drei Datensätze verarbeiten, dazu kam noch der Abbruch bei dem größten Datensatz mit zwei Workern. Im Folgenden werden die Messdaten bildlich präsentiert.

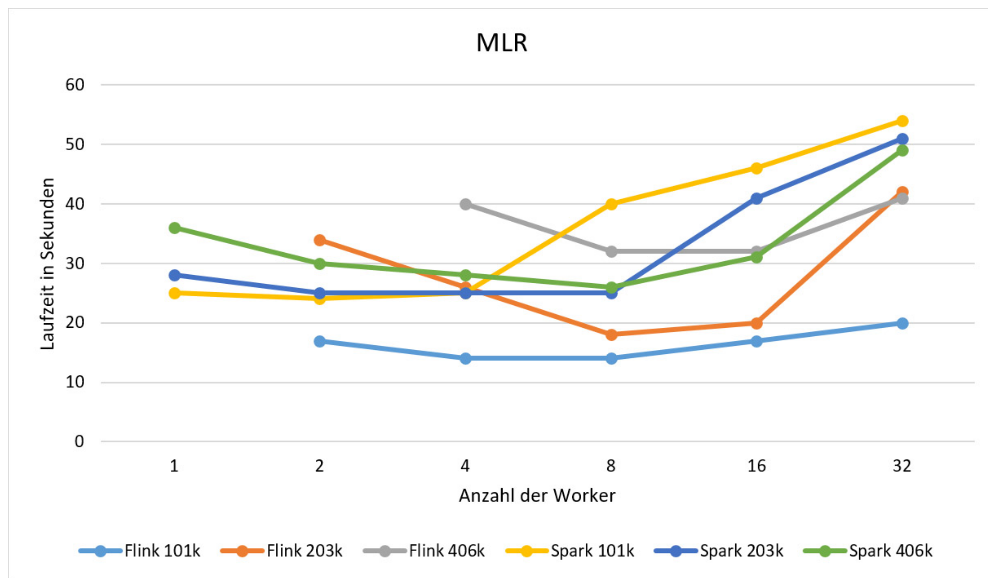

Abbildung 5.9: Multiple Linear Regression Laufzeit mit verschiedener Worker Anzahl

Auf Abbildung 5.9 sind zum ersten Mal sehr unterschiedliche Messdaten zu sehen. Hypothese „*Apache Spark und Apache Flink brauchen für die Verarbeitung einer x-mal so großen Datenmenge x-mal so lang.*“ kann für Apache Spark klar widerlegt werden, da die Messwerte zwischen 24-26 Sekunden liegen. Hingegen ist bei Apache Flink ein linearer Ansatz zu erkennen. Er liegt im Durchschnitt bei einer 1,2-fachen Laufzeiterhöhung bei der Vordoppelung des Datensatzes von 101k auf 203k und einer 1,8-fachen Laufzeiterhöhung bei der Vordoppelung des Datensatzes von 203k auf 406k. Um Hypothese „*Apache Spark und Apache Flink brauchen mit x-facher Worker Anzahl eine 1 durch x-fache Durchführungsdauer.*“ zu bestätigen müsste es mindestens bei einer Linie eine Steigung von 45 Grad im Diagramm zu sehen sein. Dies ist nicht der Fall, somit ist die Hypothese widerlegt. Eine Verschlechterung der Laufzeit ist ab der Workeranzahl 8 zu sehen; vorher verbessert sich die Laufzeit in einem geringen Ausmaß. An Hand der grauen und orangen Linie wird deutlich, dass Apache Flink beim Einsatz von mehr Workern Apache Spark im Bereich der Laufzeit überholt. Somit ist die Hypothese „*Apache Spark teilt Daten besser auf mehrere Worker auf als Apache Flink.*“ eher widerlegt als bestätigt. Hypothese „*Apache Flink arbeitet mit weniger Worker schneller im Vergleich zu Apache Spark.*“ trifft in einem von drei Fällen zu. Für die Hypothese „*Apache Flink ist bei größeren Datenmengen schneller als Apache Spark.*“ wird **??** betrachtet. Hier wurde die schnellste Zeit von Apache Flink und Apache Spark pro Datensatz gegenüber gestellt. Das Diagramm spiegelt das Gegenteil der Aussage der Hypothese wider. Bei dem großen Datensatz liegt Apache Flink hinten und bei

den anderen beiden ist es umgekehrt. Das bedeutet Apache Flink ist im ersten Fall um 71% und im zweiten Beispiel noch 39% schneller. Apache Spark hat nur im letzten Fall einen Vorsprung von 23 %.

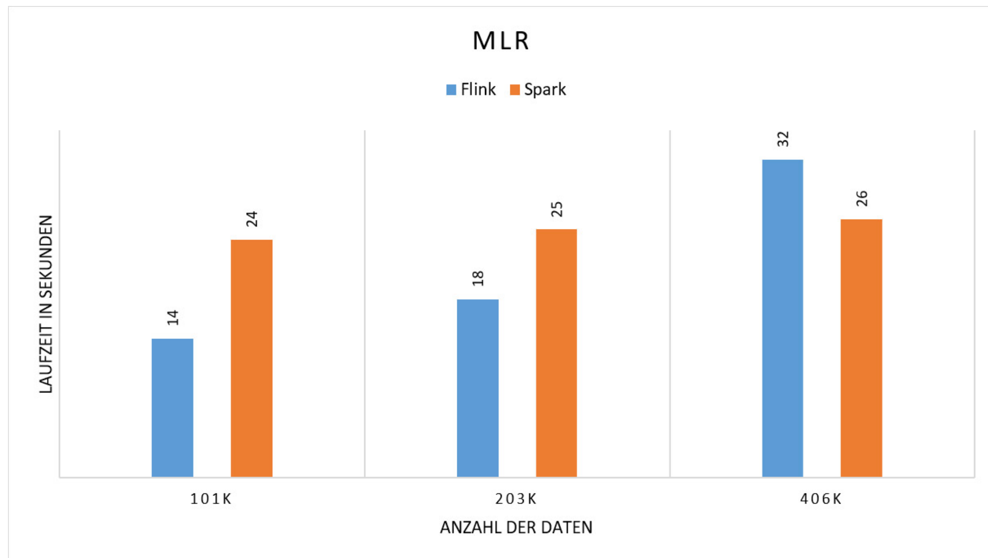

Abbildung 5.10: Multiple Linear Regression Laufzeit mit verschieden großen Datensätzen

		Worker Anzahl					
	Datensatz	1	2	4	8	16	32
Apache Flink	406k	-	-	40	32	32	41
Apache Spark	406k	36	30	28	26	31	49
Apache Flink	203k	-	34	26	18	20	42
Apache Spark	203k	28	25	25	25	41	51
Apache Flink	101k	-	17	14	14	17	20
Apache Spark	101k	25	24	25	40	46	54

Tabelle 5.7: Multiple Linear Regression Messergebnisse

5.3.6 Stochastic Outlier Selection (SOS)

Bei der Ausführung des Stochastic Outlier Selection Algorithmus wurden nur 1000 Zeilen an Daten verwendet, weil bei der Stochastic Outlier Selection jede Zeile mit allen anderen Zeilen verglichen werden muss, was einer mit sich selbst Potenzierung entspricht. Das bedeutet, dass mit einer Millionen Daten gearbeitet wird. Wie in den Versuchen zuvor gibt es auch hier Durchläufe die zu keinem Ergebnis führten. Nicht nur Apache Flink konnte die drei Datensätze mit 32 Workern nicht beenden. Keines der beiden Frameworks konnte die größeren beiden Datensätze mit einem und zwei Workern erfolgreich ausführen. Es folgt ein Diagramm, welches die Messdaten visuell darstellt.

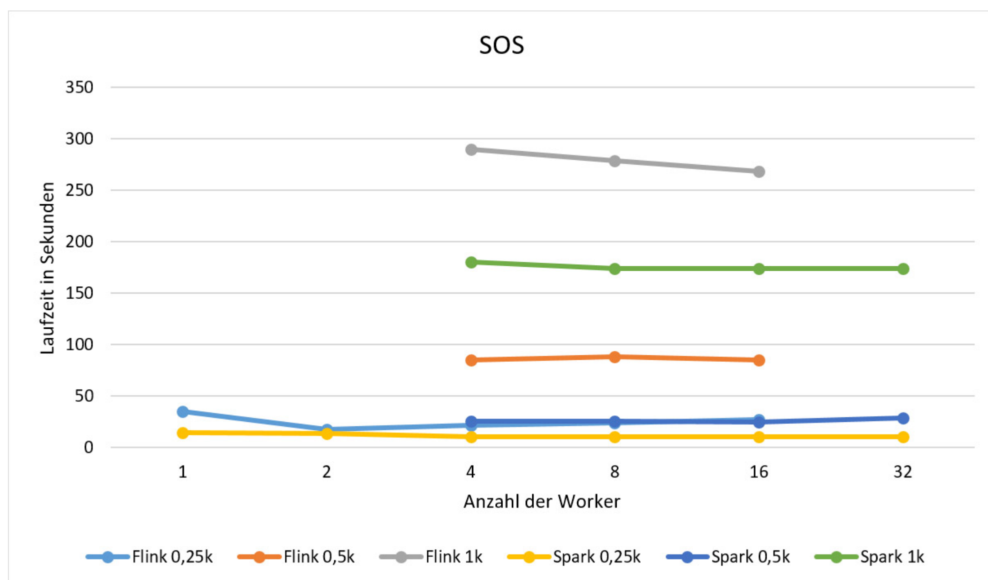

Abbildung 5.11: Stochastic Outlier Selection Laufzeit mit verschiedener Worker Anzahl

Wie in Abbildung 5.11 erstmals zu sehen ist, liegen die Linien nicht direkt aufeinander, was Hypothese *„Apache Spark und Apache Flink brauchen für die Verarbeitung einer x-mal so großen Datenmenge x-mal so lang.“* somit teilweise bestätigt. In Abbildung 5.12 ist zusehen, dass der Anstieg nicht linear verläuft, sondern bei Apache Flink bei der Verdoppelung der Daten bei 4-fachem und bei der weiteren Verdoppelung bei 3-fachem Anstieg. Wiederum ist bei Apache Spark bei der ersten Verdoppelung des Datensatzen ein linearer Anstieg zu sehen, jedoch bei der weiteren Verdoppelung der Daten ist die Laufzeit sieben Mal höher als zuvor. Durch eine waagerechte Darstellung der Linien ist Hypothese *„Apache Spark und Apache Flink brauchen mit x-facher Worker Anzahl eine 1 durch x-fache Durchführungsdauer.“* widerlegt.

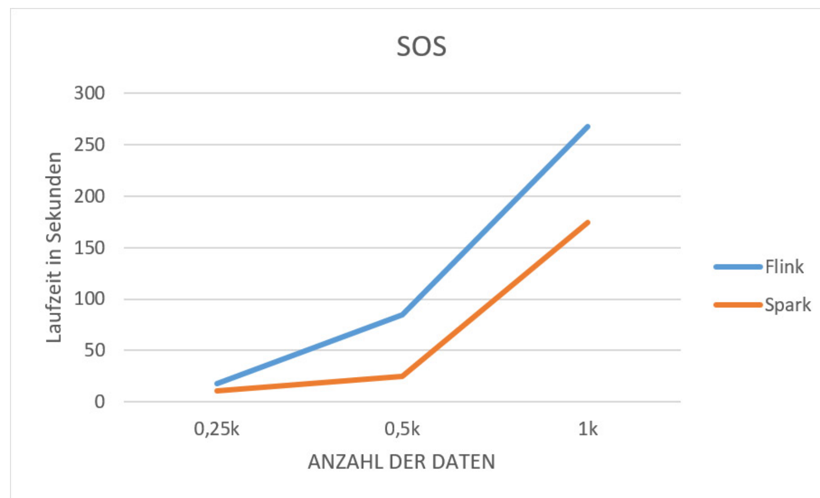

Abbildung 5.12: Stochastic Outlier Selection Laufzeit Anstieg bei verschieden großen Datensätzen

Insgesamt wurde festgestellt, dass Apache Spark bei allen Messungen schneller war, dies widerlegt Hypothese *„Apache Flink ist bei größeren Datenmengen schneller als Apache Spark.“*. Gerade bei diesem Versuch ist Apache Spark im mittleren Datensatzbereich mit 340% und im hohen Datensatzbereich mit 150% schneller als Apache Flink, wie in Abbildung 5.13 zu sehen ist. Durch diese Erkenntnis wird Hypothese *„Apache Spark teilt Daten besser auf mehrere Worker auf als Apache Flink.“* bestätigt und umgekehrt Hypothese *„Apache Flink arbeitet mit weniger Worker schneller im Vergleich zu Apache Spark.“* widerlegt.

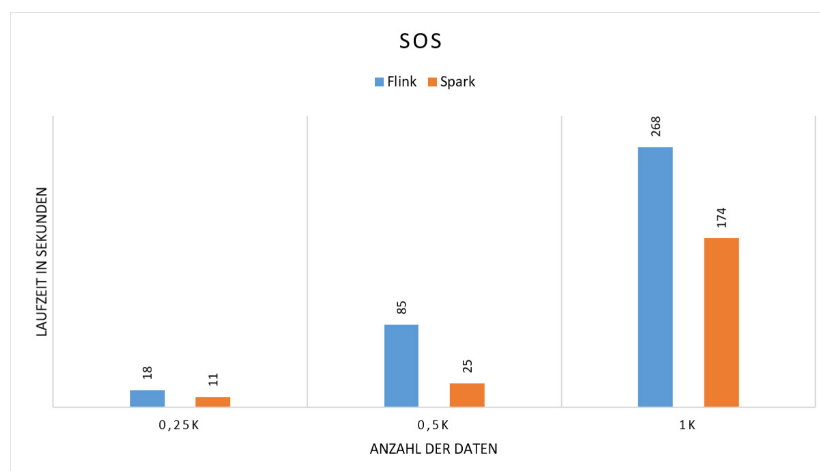

Abbildung 5.13: Stochastic Outlier Selection Laufzeit mit verschieden großen Datensätzen

		Worker Anzahl					
	Datensatz	1	2	4	8	16	32
Apache Flink	1k	-	-	290	279	268	-
Apache Spark	1k	-	-	180	174	174	174
Apache Flink	0,5k	-	-	85	88	85	-
Apache Spark	0,5k	-	-	26	26	25	29
Apache Flink	0,25k	35	18	22	24	27	-
Apache Spark	0,25k	15	14	11	11	11	11

Tabelle 5.8: Stochastic Outlier Selection Messergebnisse

5.3.7 Zusammenfassung und Bewertung der Ergebnisse

Die Ergebnisse der Experimente (5.2) variieren und hängen sehr stark von Algorithmus, Worker Anzahl und der Größe der verwendeten Daten ab. Hier konnte an Hand der Messungen bei den Algorithmen Support Vector Machine und Stochastic Outlier Selection ein klarer Vorteil in Richtung Apache Spark festgestellt werden. Hingegen zeigte Apache Flink bei den Algorithmen MinMax Scaler (wenn der Job beendet wurde) und Alternating Least Squares bessere Laufzeitergebnisse. Bei dem K-Means Clustering war Apache Spark unter Verwendung einer größeren Anzahl von Workern schneller im Vergleich. Dagegen war Apache Flink mit weniger Workern schnell und zeigte auch insgesamt die beste Laufzeit. Die größten Unterschiede konnten bei dem Algorithmus Multiple Linear Regression beobachtet werden. Hier war Apache Flink bei der Durchführung von wenig Daten in allen Messungen besser. Wiederum bei dem mittleren Datensatz lief Apache Spark mit weniger Workern schneller, jedoch bei mehr Workern war Apache Flink führend. Die insgesamt schnellste Laufzeit bei diesem Algorithmus konnte bei Apache Flink gemessen werden. Der größte Datensatz wurde von Apache Flink mit mehr Workern schneller bearbeitet, wiederum war Apache Spark mit weniger Workern schneller im Vergleich und auch insgesamt. Mit diesen Erkenntnissen lässt sich kein bestes bzw. schnellstes Framework im Bereich Machine Learning auszeichnen. Grundsätzlich ist zu sagen, dass die Cluster Konfiguration (1GB Arbeitsspeicher und Single Core CPU) eher selten bis nie für dieses Szenario eingesetzt wird. Apache Flink hatte größere Probleme mit dem zu kleinen Arbeitsspeicher für den Master Knoten, was schon bei der Ausführung mit vielen Workern aufgefallen ist. Apache Flink hat beim Einsatz von 32 Workern die Daten immer auf maximal vier teilweise auch nur auf zwei Workern verteilt, was bei zu großen Daten zum Absturz führte. Bei Apache Spark konnte nur bei der Stochastic Outlier Selection mit einem und zwei Workern, die zu kleinen Arbeitsspeicher der Slave Knoten zur nicht Beendigung des Algorithmus führen. Zu beobachten war noch die Tatsache, dass durch die Netzwerk Geschwindigkeit u.a. die Verteilung und Ausführung mit vier Workern langsamer war als die Laufzeit mit zwei Workern. Bei der Betrachtung der schnellsten Laufzeit war Apache Flink in 11 von 18 Versuchen schneller als Apache Spark.

6 Fazit

6.1 Zusammenfassung

Ziel der Arbeit war es Apache Spark und Apache Flink experimentell miteinander zu vergleichen. Hierfür wurden zunächst die Konzepte und die Architektur der beiden Frameworks beschrieben. Dabei sind Ähnlichkeiten der beiden Systeme aufgefallen. Die Teilbereiche von Machine Learning wurden erklärt und eingeordnet, worauf die Beschreibung der Besonderheiten der jeweiligen Machine Learning Bibliothek der beiden Frameworks folgt. Die Funktion der Algorithmen Support Vector Machine, MinMax Scaler, K-Means, Stochastic Outlier Selection, Multiple Linear Regression und Alternating Least Squares wurde detailliert beschrieben, um diese dann in den Experimenten auszuführen und auszuwerten. Alle Algorithmen wurden horizontal auf bis zu 32 Computern skaliert und mit verschieden großen Daten ausgeführt. Bei der Auswertung wurden Diagramme und Messdaten präsentiert und analysiert. Das Ergebnis war, das Apache Flink einen kleinen Vorteil gegenüber Apache Spark hatte. Hingegen war Apache Spark bei einigen Algorithmen schneller und Apache Flink hatte häufig Probleme bei der Ausführung auf wenigen Knoten. Die für das Experiment gewählten Hypothesen konnten alle nicht klar bestätigt oder widerlegt werden. Auch im Hinblick von schon existierenden Performance Vergleichen von Apache Spark und Apache Flink im Machine Learning Bereich wird die Wahl des Clusters, insbesondere die Wahl des sehr geringen Arbeitsspeicher deutlich, da diese Apache Spark oft im Vorteil sahen. Des Weiteren führte die fehlende Optimierung von Code und Netzwerkgeschwindigkeit zu einer Verfälschung der Ergebnisse sowie zu einer selten sichtbaren Skalierung.

6.2 Ausblick

An den Ergebnissen dieser Thesis war grundsätzlich keine Skalierbarkeit zu erkennen, um dies in künftigen Arbeiten zu veranschaulichen. Vor diesem Hintergrund erscheint es sinnvoll, wesentlich schnellere Computer und den Einsatz von größeren Arbeitsspeicher in einem Cluster einzusetzen. Es wäre zusätzlich aufschlussreich, Datensätze die im Bereich von Gigabyte oder Terabyte liegen zu untersuchen, um dem Wort Big Data gerecht zu werden. In dieser Arbeit wurde nur auf die Geschwindigkeit eingegangen. Es wäre ebenso interessant, die Fehlerquote eines Algorithmus für sich betrachtet oder in Abhängigkeit zur Geschwindigkeit in weiteren Experimenten zu überprüfen. Darüber hinaus wäre es eine Bereicherung in nachfolgenden Arbeiten herauszufinden, inwiefern die Optimierung des Algorithmus oder die Netzwerkgeschwindigkeit im Cluster einen wesentlichen Einfluss auf die Laufzeit hat. Um die Erkenntnisse auf eine breitere Basis zu stellen, wäre die Einordnung der Ergebnisse im Vergleich zu Scikit-Learn, auf denen Apache Sparks und Apache Flinks Machine Learning grundlegend basiert, sinnvoll. Es existieren viele weitere Machine Learning Frameworks für den Big Data Bereich. Diese könnten in weiteren Experimenten verglichen werden, um die besten Frameworks für bestimmte Teilgebiete zu ermitteln.

Literaturverzeichnis

[ABLT17] Assefi, M. ; Behravesh, E. ; Liu, G. ; Tafti, A. P.: Big data machine learning using apache spark MLlib. (2017), Dec, 3492-3498. http://dx.doi.org/10.1109/BigData.2017.8258338. – DOI 10.1109/BigData.2017.8258338

[BK16] Bengfort, Benjamin ; Kim, Jenny: *Data Analytics with Hadoop: An Introduction for Data Scientists*. 1st. O'Reilly Media, Inc., 2016. – ISBN 1491913703, 9781491913703

[BSRM17] Boden, Christoph ; Spina, Andrea ; Rabl, Tilmann ; Markl, Volker: Benchmarking Data Flow Systems for Scalable Machine Learning. (2017), 5:1–5:10. http://dx.doi.org/10.1145/3070607.3070612. – DOI 10.1145/3070607.3070612. ISBN 978–1–4503–5019–8

[CEF+17] Carbone, Paris ; Ewen, Stephan ; Fóra, Gyula ; Haridi, Seif ; Richter, Stefan ; Tzoumas, Kostas: State Management in Apache Flink&Reg;: Consistent Stateful Distributed Stream Processing. In: *Proc. VLDB Endow.* 10 (2017), August, Nr. 12, 1718–1729. http://dx.doi.org/10.14778/3137765.3137777. – DOI 10.14778/3137765.3137777. – ISSN 2150–8097

[CKE+15] Carbone, Paris ; Katsifodimos, Asterios ; Ewen, Stephan ; Markl, Volker ; Haridi, Seif ; Tzoumas, Kostas: Apache flink : Stream and batch processing in a single engine. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36 (2015), Nr. 4. http://sites.computer.org/debull/A15dec/issue1.htm. – QC 20161222

[Des17] Deshpande, T.: *Learning Apache Flink*. Packt Publishing, 2017. – ISBN 9781786466228

[DXS+15] Dudoladov, Sergey ; Xu, Chen ; Schelter, Sebastian ; Katsifodimos, Asterios ; Ewen, Stephan ; Tzoumas, Kostas ; Markl, Volker: Optimistic Recovery for

Iterative Dataflows in Action. (2015), 1439–1443. http://dx.doi.org/10.1145/2723372.2735372. – DOI 10.1145/2723372.2735372. ISBN 978-1-4503-2758-9

[Fli19a] FLINK, Apache: *Apache Flink 1.7 Documentation: Alternating Least Squares*. https://ci.apache.org/projects/flink/flink-docs-release-1.7/dev/libs/ml/als.html. Version: 2019

[Fli19b] FLINK, Apache: *Apache Flink 1.7 Documentation: Component Stack*. https://ci.apache.org/projects/flink/flink-docs-release-1.7/internals/components.html. Version: 2019

[Fli19c] FLINK, Apache: *Apache Flink 1.7 Documentation: Gelly: Flink Graph API*. https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/gelly/index.html. Version: 2019

[Fli19d] FLINK, Apache: *Apache Flink 1.7 Documentation: Iterations*. https://ci.apache.org/projects/flink/flink-docs-stable/dev/batch/iterations.html. Version: 2019

[Fli19e] FLINK, Apache: *Apache Flink 1.7 Documentation: Looking under the hood of pipelines*. https://ci.apache.org/projects/flink/flink-docs-release-1.7/dev/libs/ml/pipelines.html. Version: 2019

[Fli19f] FLINK, Apache: *Apache Flink 1.7 Documentation: MinMax Scaler*. https://ci.apache.org/projects/flink/flink-docs-release-1.7/dev/libs/ml/min_max_scaler.html. Version: 2019

[Fli19g] FLINK, Apache: *Apache Flink 1.7 Documentation: Quickstart Guide*. https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/ml/quickstart.html. Version: 2019

[Fli19h] FLINK, Apache: *Apache Flink 1.7 Documentation: Stochastic Outlier Selection*. https://ci.apache.org/projects/flink/

flink-docs-release-1.7/dev/libs/ml/sos.html. Version: 2019

[Fli19i] FLINK, Apache: *Apache Flink 1.7 Documentation: Table API and SQL*. https://ci.apache.org/projects/flink/flink-docs-release-1.7/dev/table/. Version: 2019

[Fli19j] FLINK, Apache: *Apache Flink: What is Apache Flink?* https://flink.apache.org/flink-applications.html. Version: 2019

[FSW16] FU, J. ; SUN, J. ; WANG, K.: SPARK - A Big Data Processing Platform for Machine Learning. (2016), Dec, 48-51. http://dx.doi.org/10.1109/ICIICII.2016.0023. – DOI 10.1109/ICIICII.2016.0023

[GGRGGH17] GARCÍA-GIL, Diego ; RAMÍREZ-GALLEGO, Sergio ; GARCÍA, Salvador ; HERRERA, Francisco: A comparison on scalability for batch big data processing on Apache Spark and Apache Flink. In: *Big Data Analytics* 2 (2017), Mar, Nr. 1, 1. http://dx.doi.org/10.1186/s41044-016-0020-2. – DOI 10.1186/s41044–016–0020–2. – ISSN 2058–6345

[Gro16] GROUPLENS: *MovieLens 20M Dataset*. https://grouplens.org/datasets/movielens/. Version: 2016

[GW17] GUI, J. ; WANG, Q.: Topic modeling of news based on spark Mllib. (2017), 224-228. http://dx.doi.org/10.1109/ICCWAMTIP.2017.8301484. – DOI 10.1109/ICCWAMTIP.2017.8301484

[KKWZ15] KARAU, Holden ; KONWINSKI, Andy ; WENDELL, Patrick ; ZAHARIA, Matei: *Learning Spark: Lightning-Fast Big Data Analytics*. 1st. O'Reilly Media, Inc., 2015. – ISBN 1449358624, 9781449358624

[Rep10] REPOSITORY, UC Irvine Machine L.: *Individual household electric power consumption Data Set*. http://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption. Version: 2010

[SDC+16] SALLOUM, Salman ; DAUTOV, Ruslan ; CHEN, Xiaojun ; PENG, Patrick X. ; HUANG, Joshua Z.: Big data analytics on Apache Spark. In: *International Journal of Data Science and Analytics* 1 (2016), Nov, Nr. 3, 145–164. http://dx.doi.org/

10.1007/s41060-016-0027-9. – DOI 10.1007/s41060–016–0027–9. – ISSN 2364–4168

[SGA16] Siegal, D. ; Guo, J. ; Agrawal, G.: Smart-MLlib: A High-Performance Machine-Learning Library. (2016), Sep., S. 336–345. http://dx.doi.org/10.1109/CLUSTER.2016.49. – DOI 10.1109/CLUSTER.2016.49. – ISSN 2168–9253

[Spa18a] Spark, Apache: *Cluster Mode Overview - Spark 2.4.0 Documentation.* https://spark.apache.org/docs/latest/cluster-overview.html. Version: 2018

[Spa18b] Spark, Apache: *GraphX - Spark 2.4.0 Documentation.* https://spark.apache.org/docs/latest/graphx-programming-guide.html. Version: 2018

[Spa18c] Spark, Apache: *ML Pipelines - Spark 2.4.0 Documentation.* https://spark.apache.org/docs/latest/ml-pipeline.html. Version: 2018

[Spa18d] Spark, Apache: *MLlib: Main Guide - Spark 2.4.0 Documentation.* https://spark.apache.org/docs/latest/ml-guide.html. Version: 2018

[Spa19a] Spark, Apache: *Apache Spark™ is a unified analytics engine for large-scale data processing.* https://spark.apache.org/. Version: 2019

[Spa19b] Spark, Apache: *Spark SQL and DataFrames - Spark 2.4.0 Documentation.* https://spark.apache.org/docs/latest/sql-programming-guide.html. Version: 2019

[Vas16] Vasiloudis, Theodore: *FlinkML: Vision and Roadmap - Apache Flink - Apache Software Foundation.* https://cwiki.apache.org/confluence/display/FLINK/FlinkML%3A+Vision+and+Roadmap. Version: 2016

[VTK16] Verbitskiy, I. ; Thamsen, L. ; Kao, O.: When to Use a Distributed Dataflow Engine: Evaluating the Performance of Apache Flink. (2016), July, 698-705. http://dx.doi.org/10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.0114. – DOI 10.1109/UIC–ATC–ScalCom–CBDCom–IoP–SmartWorld.2016.0114

[ZXW+16] Zaharia, Matei ; Xin, Reynold S. ; Wendell, Patrick ; Das, Tathagata ; Armbrust, Michael ; Dave, Ankur ; Meng, Xiangrui ; Rosen, Josh ; Venkataraman, Shivaram ; Franklin, Michael J. ; Ghodsi, Ali ; Gonzalez, Joseph ; Shenker, Scott ; Stoica, Ion: Apache Spark: A Unified Engine for Big Data Processing. In: *Commun. ACM* 59 (2016), Oktober, Nr. 11, 56–65. `http://dx.doi.org/10.1145/2934664`. – DOI 10.1145/2934664. – ISSN 0001–0782

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 25. Juni 2019 Jannik Bruhns