HAW HAMBURG

BACHELORTHESIS
Finn Masurat

Realtime-Erkennung von Hate-Speech auf Twitter: Eine Evaluation von Apache Flink und Spark

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Finn Masurat

Realtime-Erkennung von Hate-Speech auf Twitter: Eine Evaluation von Apache Flink und 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. Stefan Sarstedt

Eingereicht am: 20. Dezember 2019

Finn Masurat

Thema der Arbeit

Realtime-Erkennung von Hate-Speech auf Twitter: Eine Evaluation von Apache Flink und Spark

Stichworte

Apache Flink, Apache Spark, Twitter, Echtzeit, Filter, Datenströme, Support Vector Machine, Maschinelles Lernen, Klassifizierung

Kurzzusammenfassung

Apache Flink und Apache Spark bieten beide eine Schnittstelle für Streaming Anwendungen sowie Bibliotheken des maschinellen Lernens an. Im Rahmen dieser Thesis soll verglichen werden, welches Framework sich besser für den Gebrauch als Realtime Hate-Speech Filter auf Twitter eignet.

Finn Masurat

Title of Thesis

Realtime detection of Hate-Speech on Twitter: An evaluation of Apache Flink and Spark

Keywords

Apache Flink, Apache Spark, Twitter, Real-Time, Filter, streaming, support vector machine, machine learning, classification

Abstract

Apache Flink and Apache Spark both provide an interface for streaming applications and machine learning libraries. This thesis will compare which framework is better suited for use as a real-time hate speech filter on Twitter

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

Der Ausdruck Hate-Speech ist in der gegenwärtigen Zeit ständiges Begleitwort, wenn es um die sozialen Medien geht. Im aktuellen Fall der Hassrede auf Facebook gegen die österreichische Ex-Politikerin Eva Glawischnig kam der Europäische Gerichtshof (EuGH) am dritten Oktober 2019 zu dem Urteil, dass hetzerische Beiträge gegen diese entfernt werden müssen. Doch selbst wenn einzelne Beiträge durch die Betreiber der sozialen Medien entfernt werden, verbreiten sich in kürzester Zeit wortidentische oder ähnliche Beleidigungen durch weitere Nutzer der Plattformen. Für die Lösung dieses Problems ist eine automatische Erkennung der Hassrede unabdingbar, denn jede Sekunde werden beispielsweise sechstausend Nachrichten auf Twitter veröffentlicht. Dies umfasst eine Größe an Nachrichten, welche von Menschenhand nicht mehr kontrollierbar ist.

1.1 Ziel der Bachelorarbeit

Das Ziel dieser Bachelorarbeit ist die Evaluierung der Frameworks Apache Flink und Apache Spark im Bezug auf die Erstellung einer Realtime Hate-Speech Erkennung. Dafür werden beide Frameworks in den für dieses Vorhaben wichtigen Punkten geprüft. Dazu gehören die Text-Vorverarbeitung, die Klassifizierung von Texten, die Einfachheit der Implementierungen selbst, sowie die Fähigkeit Daten durch Datenströme schnell zu erfassen und auszuwerten. Dabei liegt der Fokus auf der technischen Leistungsfähigkeit beider Frameworks, diese Art der Klassifizierung durchzuführen, nicht auf den Klassifizierungsresultaten.

1.2 Aufbau der Bachelorarbeit

Kapitel 2 beschäftigt sich zunächst mit dem Begriff Hate-Speech, der Support Vector Machine und der Twitter API.

Im Kapitel 3 und 4 werden die Grundlagen der Frameworks Apache Flink und Apache Spark erklärt.
Kapitel 5 stellt wissenschaftliche Artikel und deren herausgearbeitete Erkenntnisse vor, welche sich ebenfalls mit dem Vergleich von Apache Flink und Apache Spark beschäftigen. Im Kapitel 6 werden die Implementationen der beiden Realtime Hate-Speech-Filter Anwendungen vorgestellt.
Im Kapitel 7 werden die Leistungen der beiden Frameworks durch Experimente getestet.
Im Kapitel 8 wird die Thesis nochmals zusammengefasst und es werden mögliche Arbeiten zum Thema Realtime Hate-Speech Erkennung vorgestellt, welche auf dieser Thesis aufbauen könnten.

2 Hate-Speech, Support Vector Machine und die Twitter API

Dieses Kapitel dient der Klärung des Begriffes *Hate-Speech* und gibt anschließend einen Überblick über das im späteren Versuch benutzte Verfahren zum Filtern von Hate-Speech. Ebenso wird die Twitter API vorgestellt, welche als Datenlieferant für die zu klassifizierenden Tweets dient.

2.1 Hate-Speech

Was ist *Hate-Speech*? - Diese Frage zu beantworten und zu entscheiden, ob ein Text Hate-Speech enthält, ist für den menschlichen Verstand nicht immer eindeutig zu lösen. Hate-Speech (dt. Hassrede) ist ein komplexes Phänomen, welches eng mit den Beziehungen zwischen Gruppen verbunden ist und sich manchmal nur auf Sprachnuancen auswirkt [Was16]. Es ist also umso wichtiger, Hate-Speech klar zu definieren, um ihre im späteren Verlauf implementierte Identifizierung zu erleichtern [RRC+17].

„Hass schürendes Verhalten: Du darfst keine Gewalt gegen andere Personen fördern, insbesondere nicht aufgrund der Abstammung, der ethnischen Zugehörigkeit, der nationalen Herkunft, der sexuellen Orientierung, des Geschlechts, der Geschlechtsidentität, der religiösen Zugehörigkeit, des Alters, einer Behinderung oder einer Krankheit, noch darfst du sie aus diesen Gründen bedrohen oder belästigen.“ [Twi19b]

„Sprache, die eine Gruppe angreift oder erniedrigt, die auf Rasse, ethnischer Herkunft, Religion, Behinderung, Geschlecht, Alter oder sexueller Orientierung/Geschlechtsidentität beruht.“ [NTT+16]

„Wir lassen Humor und Gesellschaftskritik in Verbindung mit diesen Themen zu.“ [Fac19]

Die allgemeinen Definitionen von Hate-Speech haben, wie die obigen zeigen, oft Überschneidungen im Inhalt. Die erste Definition ist von Twitter selbst im Regelwerk definiert, die zweite Definition aus einer wissenschaftlichen Arbeit. Das dritte Zitat stammt aus den Facebook Gemeinschaftsstandards und zeigt deutlich die Schwierigkeit der Definition von Hate-Speech. Denn was für eine Gruppe Humor oder Satire sein kann, kann für eine andere Gruppe anstößig oder erniedrigend sein.

Fortuna und Nunes [FN18] werteten in ihrer wissenschaftlichen Arbeit verschiedene Auslegungen von Hate-Speech aus und erstellten aus den Resultaten ihrer Forschung eine eigene Definition.
Diese Definition ist auch Grundlage dieser Thesis und beschreibt Hate-Speech als angreifende oder erniedrigende Sprache. Eine Sprache, welche Gewalt oder Hass gegen Gruppen hervorruft, basierend auf spezifischen Merkmalen wie körperlicher Erscheinung, Religion, Abstammung, nationaler oder ethnischer Herkunft, sexueller Orientierung, Geschlechtsidentität oder anderem. Ebenso kann diese Art der Sprache mit verschiedenen Sprachstilen auftreten, selbst in subtilen Formen oder wenn Humor verwendet wird.

2.2 Support Vector Machine

Support Vector Machine, oder auch abgekürzt SVM, ist ein Verfahren des überwachten Lernens im maschinellen Lernen, welches mit Hilfe von Klassifikation und Reggression Daten analysiert. Dieses Verfahren wird in dem Versuchsaufbau in Kapitel 6 zum Erlernen der Features der Trainingsdaten und der Klassifizierung der Tweets verwendet, weswegen in diesem Kapitel näher darauf eingegangen wird.

SVM ist eine sehr leistungsfähige Lernmethode, die viele Probleme des maschinellen Lernens, welche mit der Effizienz des Trainings in Verbindung stehen, wie beispielsweise *Overfitting*, überwindet. Eine SVM kann sich gut an hochdimensionale Daten, in denen umfangreiche Darstellungen von Wörtern, *Multigrammen*, etc. enthalten sind, anpassen. So ist sie im Stande, auch bei Trainingssets mit vielen Tausenden von Beispielen effiziente Lösungen zu finden [GS04].

Entwickelt wurde die Support Vector Machine von Corinna Cortes und Vladimir Vapnik [CV95] für die binäre Klassifizierung. Die Grundkonzepte ihres Ansatzes lassen sich grob unter folgenden Punkten zusammenfassen:

Klassentrennung: Ziel ist es, die optimal trennende Hyperebene (*hyperplane*) zwischen zwei Klassen zu finden, indem der Abstand (*margin*) zwischen den engsten Punkten der beiden Klassen maximiert wird. Veranschaulicht ist dies in Abbildung 2.1, in der die beiden Klassen klar voneinander getrennt werden können. Die Punkte an der Grenze werden *Support Vectors* genannt. Die Mitte der Spanne zwischen diesen ist die optimal trennende Hyperebene.

Klassenüberschneidung: Sind einzelne Punkte einer Klasse auf der „falschen" Seite der *Diskriminanzfunktion* [DRB10], werden diese weniger gewichtet, um ihren Einfluss auf die Berechnung zu reduzieren. Dieses Vorgehen wird auch „Soft Margin" genannt.

Nichtlinearität: Kann keine lineare Trennung der beiden Klassen gefunden werden, transformiert man die Datenpunkte in einen höherdimensionalen Raum, in dem man sich eine bessere lineare Separierbarkeit erhofft. Diese Transformation geschieht durch die Kernel-Funktion.
Aufgaben werden beispielsweise als quadratisches Optimierungsproblem dargestellt, wodurch sie mit bekannten Algorithmen gelöst werden können.
(Siehe Abbildung 2.2)

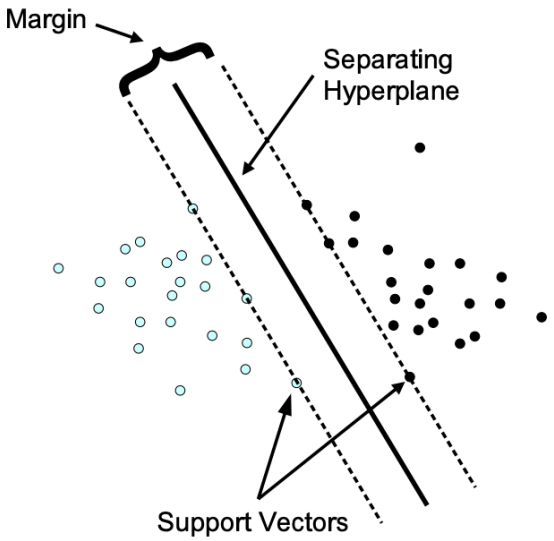

Abbildung 2.1: SVM - Klassifizierung *(Quelle: [MW15])*

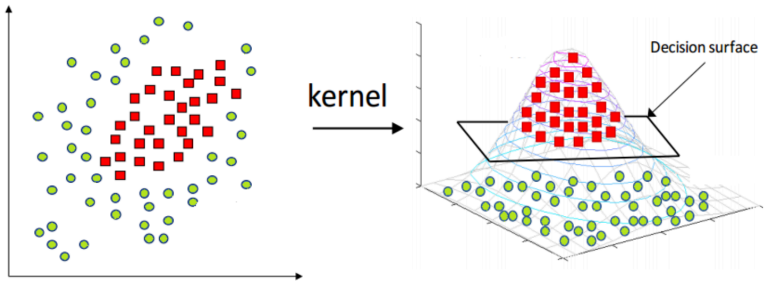

Abbildung 2.2: SVM - Kernelfunktion *(Quelle: https://www.hackerearth.com/)*

Ein Programm, das in der Lage ist, all diese Schritte auszuführen, wird als Support Vector Machine bezeichnet [MW15].

2.2.1 Begründung der Wahl der SVM

Für das Klassifizieren von Texten existieren eine Reihe von Algorithmen des maschinellen Lernens.
Neben der SVM ist der am häufigsten verwendete Algorithmus der *Naive Bayes* Algorithmus. Dieser Algorithmus, welcher auf dem *Bayes* Theorem basiert, punktet durch seine Einfachheit und die Möglichkeit, schnelle Adaptionen bei falsch klassifizierten Texten durchzuführen. Der Begriff *„naive“* kommt von der Art und Weise, wie der Algorithmus die einzelnen Feature, in diesem Falle Wörter, in Beziehung setzt, um die Sätze zu klassifizieren. Es wird angenommen, dass der Wert eines Wortes unabhängig von dem Wert jedes anderen Wortes ist. Zum Beispiel könnte eine Frucht als Apfel betrachtet werden, wenn diese rot, rund und etwa 10cm im Durchmesser ist. Ein Naive Bayes Klassifikator würde diese drei Merkmale unabhängig voneinander, ungeachtet möglicher Korrelationen zwischen den Merkmalen Farbe, Rundheit und Größe, als Beitrag zur Wahrscheinlichkeit betrachten, dass diese Frucht ein Apfel ist.

Die SVM, der Naives Bayes und andere Algorithmen wurden in verschiedenen Artikeln auf Basis der Genauigkeit der Klassifizierungen verglichen. Das Ergebnis war eine Überlegenheit der SVM gegenüber den anderen Algorithmen [TSSN15]. Deshalb wurde für diese Thesis entschieden, die SVM als Klassifikator zu verwenden.

2.3 Twitter API

Twitter bietet Entwicklern verschiedene Schnittstellen, um mit Twitter und dessen Tweets zu arbeiten. Die *Twitter Developer Platform* beinhaltet jegliche Hilfsmittel, um beispielsweise Anwendungen mit Twitter-Daten zu erstellen, Werbekampagnen auf Twitter zu gestalten oder Twitter in die eigenen Anwendungen zu integrieren.
Dafür stellt Twitter viele verschiedene APIs bereit, welche die oben genannten Felder abdecken. Die kostenlose Standard API beinhaltet *REST APIs* und Streaming APIs. Ebenso gibt es noch eine Premium und Enterprise API. Diese beiden kostenpflichtigen APIs verbinden die Standard API mit hilfreichen Filtern, historischer Suche und mehr, um tiefere Datenanalyse zu betreiben [Twi19a].

Um mit der in diesem Versuch benötigten Twitter Streaming API zu arbeiten, benötigt man einen Twitter-Developer Account, über den man sich eine Twitter-App erstellen muss. Jede Twitter-App besitzt individuelle Consumer-Keys und Access-Tokens. Für die Verbindung muss ein HTTP Request mit den Keys und Tokens an die *TwitterSource* gesendet werden. Ist die Verbindung hergestellt, kann die Response so lang wie erwünscht verarbeitet werden. Die Twitter-Server halten die Verbindung auf unbestimmte Zeit offen.
Tweets werden als *JSON-Objekt* kodiert von der TwitterSource zurückgegeben. JSON basiert auf Schlüssel-Werte-Paaren mit benannten Attributen und zugehörigen Werten. Diese Attribute und ihr Zustand werden zur Beschreibung von Objekten, in diesem Falle Tweets, verwendet. Jeder Tweet besitzt einen Autor, eine Nachricht, eine ID, einen Zeitstempel, wann der Tweet veröffentlicht wurde und geographische Metadaten. Ebenso als JSON-Objekt wird der User mit der Anzahl seiner *Follower*, seines Twitter Namens und meist einer Account Biographie mitgeschickt [Twi19a]. So kann der Tweet anhand bestimmter Attribute, wie beispielsweise dem Herkunftsland, gefiltert werden.

3 Apache Flink

Apache Flink [Fli19] ist ein in Java und Scala geschriebenes Open-Source Stream Processing Framework, welches Datenanalyse auf Basis von Datenströmen und Batch-Jobs ermöglicht. Entstanden aus dem Berliner Forschungsprojekt Stratosphere [ABE+14], wird es seit der Übernahme im Jahr 2014 durch die Apache Software Foundation als Top-Level Projekt weiterentwickelt. Die Grundidee ist eine native Datenstromverarbeitung. Anwendungen wie Echtzeitananalysen, historische Datenverarbeitungen (Batch) und iterative Algorithmen (z.B. maschinelles Lernen) sollen innerhalb einer Pipeline als fehlertolerante Datenströme ausgedrückt und ausgeführt werden können [CKE+15]. Die Arbeit mit Datenströmen bildet die Standardimplementierung, Batch-Jobs werden als Speziallfälle von Streaming-Anwendungen ausgeführt.

In diesem Kapitel wird zunächst die Systemarchitektur (3.1) von Apache Flink vorgestellt. Danach wird erklärt, wie die Streaming-Verarbeitung bei Apache durchgeführt wird (3.2). Zum Schluss wird das maschinelle Lernen (3.3) und insbesondere die Support Vector Machine Implementierung von Apache Flink (3.3.1) präsentiert.

3.1 Systemarchitektur

Die Systemarchitektur von Flink lässt sich trotz wachsender Anzahl an APIs in vier Ebenen unterteilen: Das Deployment, der Kern, die APIs und die Bibliotheken. Wie Abbildung 3.1 zeigt, bildet die Runtime den Kern von Flink. Dazu bietet Flink zwei APIs an, die DataStream API zum Verarbeiten von potenziell endlosen Datenströmen (stream processing) und die DataSet API zum Verarbeiten von begrenzten Datensätzen (batch processing). Die Runtime agiert also als eine verteilte Verarbeitungsmaschine, welche als gemeinsame Struktur dient, um sowohl die gebundene (Batch-) als auch die unbegrenzte (Stream-)Verarbeitung zu abstrahieren und zustandsbehaftete Berechnungen über begrenzte oder unbegrenze Datenströme auszuführen. Die beiden APIs erzeugen für diese Verarbeitungsmaschine die jeweiligen ausführbaren Anwendungen. Zusätzlich zu den

Kern-APIs bündelt Flink domänenspezifische Bibliotheken und APIs, welche wieder DataSet API oder DataStream API Anwendungen generieren. Zur Zeit bestehen diese aus FlinkML für das maschinelle Lernen, Gelly für die Grafikverarbeitung, die Table API für SQL-ähnliche Operationen und FlinkCEP, womit beispielsweise Ereignismuster in einem endlosen Strom aus Ereignissen erkannt werden können [CKE+15].

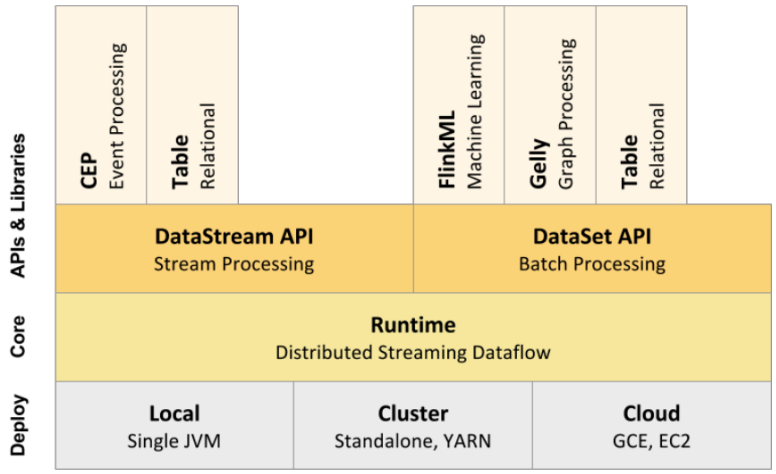

Abbildung 3.1: Flink Systemarchitektur:Komponenten
(Quelle: https://flink.apache.org)

Ein Flink Programm kann lokal, im Cluster-Modus oder in der Cloud ausgeführt werden. Bei der lokalen Ausführung wird die Flink Anwendung in einer Java VM ausgeführt. Werden größere Datenmengen mit verteilten Maschinen benutzt, kann das Programm im Cluster-Modus mit Systemen wie Hadoop YARN und Mesos, welche nativ verfügbar sind, ausgeführt werden. Im Cloud-Modus können die Google-Cloud und die Amazon-Cloud eingebunden werden [KZ18].

Flink ermöglicht eine hohe Fehlertoleranz durch die Verwendung eines Snapshot-Mechanismus, welcher konsistente Momentaufnahmen(engl. Snapshot) des verteilten Datenstoms und des Betriebszustandes erzeugt. So kann im Fehlerfall das System auf diese Momentaufnahmen zurückgreifen und die Operatoren erneut starten lassen.

3.2 Streaming Verarbeitung in Flink

Flink's Runtime ist auf die Verarbeitung unbegrenzter Datenströme optimiert und ermöglicht es, die Berechnungen auf Tausenden von Kernen zu skalieren. Dies sorgt für einen hohen Durchsatz mit niedriger Latenz.

Da man auf Datenströmen, welche unendlich lang sind, keine Operationen wie z.B. „zähle alle Elemente“, ausführen kann, bedient sich Flink Fenstern, welche den Datenstrom unterteilen. Generell können diese Fenster zeit- oder datengesteuert sein, wodurch Aufgaben wie „addiere alle Elemente der letzten 5 Minuten“oder „addiere die letzten 100 Elemente “möglich sind.
Wenn man sich in diesem Streaming-Kontext auf Zeit bezieht, um genannte Fenster zu definieren, kann es sich um verschiedene Zeitvorstellungen handeln. Die Event Time bezieht sich auf die Zeit, in der das Event erstellt wurde. Generell wird diese Zeit durch einen Zeitstempel von dem Service, welcher das Event erstellt hat, in dem Event selbst beschrieben. Die Ingestion Time beschreibt den Eintritt des Events in den Flink-Datenfluss beim Source-Operator. Zuletzt gibt es noch die Processing Time, welche die lokale Zeit der Verarbeitung der einzelnen Operatoren wiedergibt [Fli19].

Durch die angebotene DataStream API kann der Benutzer Ereignisse aus einem oder mehreren Datenströmen verarbeiten, welche aus verschiedenen Quellen, wie z.B. Message Queues, Socket Streams oder Dateien, erstellt werden können. Auf Basis der Ereignisse dieser Datenströme, werden Berechnungen durchgeführt, deren Ergebnisse in sogenannten Sinks (dt. Senke) zurückgegeben werden. Dies kann die einfache Konsolenausgabe, das Schreiben der Ergebnisse in eine Datei oder das Speichern in eine Datenbank beinhalten. Außerdem können diese Ergebnisse selbst wieder als Quelle für neue Berechungen dienen, wodurch ein Datenfluss aus Input- und Output-Streams entsteht, bis der gewollte Zustand wieder in einem Sink endet. Allgemein lässt sich damit ein Flink Programm in fünf Schritte unterteilen:

1. Beschaffung einer Ausführungsumgebung - dies ist die StreamExecutionEnvironment, welche als Basis für alle Flink Programme dient.
2. Das Laden oder Erstellen der Ausgangsdaten.
3. Bestimmung der Berechnungen oder Transformationen für die eingehenden Daten.
4. Bestimmung des Zielortes der Berechnungsresultate.

5. Ausführung des Programmes.

In der folgenden Darstellung 3.2 sieht man diesen Programmfluss, in dem eine Quelle (Source) über einen Datenstrom zur ersten Berechnung führt. Der Map-Operator wird ausgeführt und erzeugt wiederum einen Datenstrom, aus dem eine Berechnung resultiert. Nach Abschluss dieser endet es in dem zuvor erklärten Sink [Fli19].

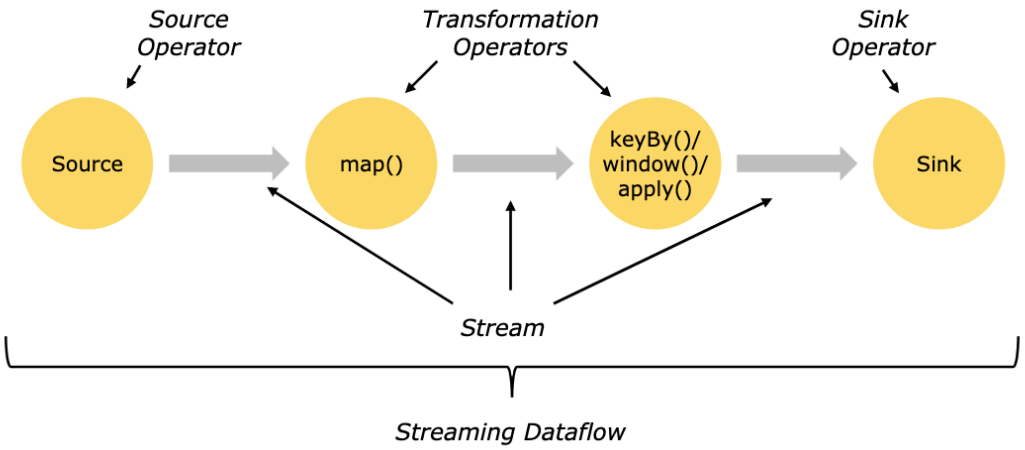

Abbildung 3.2: Flink Streaming Dataflow *(Quelle: https://flink.apache.org)*

Dieser einfache Streaming-Dataflow ist die komprimierte Darstellung einer Aufgabe. Während der Ausführung hat ein Datenstrom meist eine oder mehrere Stream-Partitionen und ein Operator ein oder mehrere Subtasks, welche unabhängig voneinander in verschieden Threads oder Computern ausgeführt werden. Dies ist speziell bei größeren Datenströmen essentiell, da der einfache Prozess hier nicht mehr genügt, um die großen Datenmengen zu verarbeiten. Zur Koordination dieser Datenströme benötigt deshalb jede Flink Applikation zur Laufzeit mindestens einen oder mehrere TaskManager sowie mindestens einen oder mehrere Job-Manager. Letztere koordinieren die verteilte Ausführung der Applikation auf die einzelnen TaskManager und kümmern sich um die Wiederherstellung bei Ausfällen. TaskMananger hingegen, welche auch Worker genannt werden, führen die ihnen zugewiesenen Tasks aus, tauschen die Datenströme und Zwischenergebnisse untereinander aus und liefern in regelmäßigen Zeitabständen einen Status an den JobManager.

Die Darstellung 3.3 zeigt nun den gleichen Prozess wie Darstellung 3.2, jedoch dieses Mal mit den Subtasks. An diesem Beispiel sieht man die Wichtigkeit des Map-Operators, da dieser die jeweiligen Daten der Subtasks miteinander verbindet [Fli19].

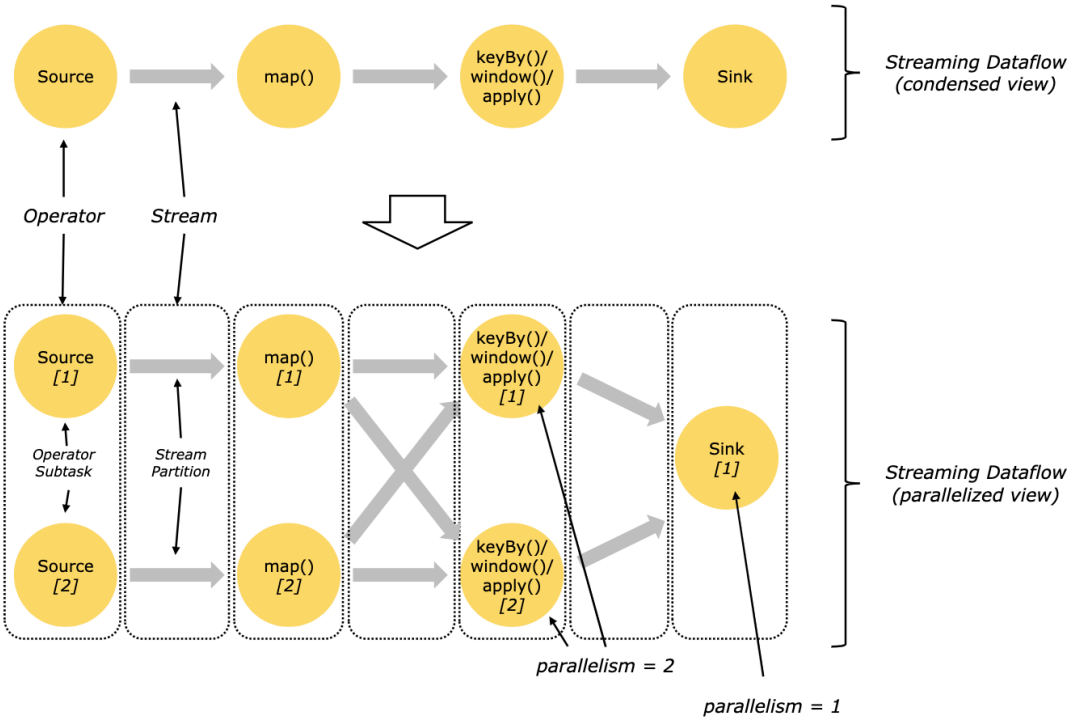

Abbildung 3.3: Flink Parallel Streaming Dataflow *(Quelle: https://flink.apache.org)*

3.2.1 DataSet API

Die DataSet API ist ein Spezialfall der Flink Anwendung. Theoretisch betrachtet handet es sich bei der DataSet API um einen Stream, denn die Runtime stellt alle Flink Anwendungen als einen Streaming Dataflow dar. Um dennoch herkömmliche Batch-Verarbeitung zu ermöglichen, teilt Flink den potentiell unendlichen Datenstrom in abgeschlossene Fenster ein [TRH+15].

Dies ist nötig, da reine Stream-Verarbeitungssysteme nicht immer die beste Wahl sind, wenn es um begrenzte Datenströme geht. Die Einführung von Streamprozessoren hat die Batch-Prozessoren nicht überflüssig gemacht. So sind z.B. Stream-Verarbeitungssysteme bei Batch-Verarbeitungs-Workloads sehr langsam. Niemand würde es für eine gute Idee halten, einen Stream-Prozessor zu verwenden, welcher durch Message-Queues schleicht,

um große Mengen an verfügbaren Daten zu analysieren [SE19]. Flink hat eine Streaming API, welche begrenzte und unbegrenzte Anwendungsfälle durchführen kann, bietet aber dennoch eine seperate DataSet API und Runtime Stack, welche für Batch-Verarbeitungen schneller sind.

Da es sich um einen Spezialfall des Streamings handelt, gelten die Konzepte der Streaming-Programme ebenso für die Batch-Programme. Jedoch gibt es kleine Ausnahmen. So verwenden Batch-Programme keinen Snapshot-Mechanismus für die Fehlertoleranz. Fehlerhafte oder abgebrochene Stream-Partitionen werden wiederholt. Dies ist möglich, da die Eingänge begrenzt sind. Die Kosten der Wiederherstellung steigen dadurch, jedoch macht dies die reguläre Verarbeitung kostengünstiger, da Kontrollpunkte vermieden werden.

3.3 Machine Learning in Flink

Flink ML ist die Bibliothek für maschinelles Lernen. Sie verwendet die DataSet API und wird durch die Flink Community vorangetrieben. Die Bibliothek besitzt eine wachsende Anzahl an Algorithmen und Mitwirkenden. Ziel ist es, skalierbare Algorithmen des maschinellen Lernens zur Verfügung zu stellen. Dabei soll sogenannter „Glue Code“, also Programmcode, welcher rein zur Verbindung normalerweise nicht kompatibler Teile des Programmcodes dient, vermieden werden. Skalierbare Extraktionen, Transformationen und Laden der Daten ist in dem Ecosystem von Flink im selben Programm möglich. So lassen sich Pipelines ohne das Einbinden anderer Technologien realisieren. Flink ML unterstürzt Algorithmen des unüberwachten Lernens, der Datenvorverarbeitung, der Empfehlung (Recommendation), sowie des überwachten Lernens, welches für den späteren Versuchsaufbau benötigt wird.

3.3.1 Support Vector Machine Implementierung in Flink

Flink bietet eine native Implementierung der Support Vector Machine an und verwendet dafür den *Communication-efficient distributed dual Coordinate Ascent (CoCoA)* Algorithmus. Dieser Algorithmus unterstützt Ziele für die linear regulierte Verlustminimierung. CoCoA kombiniert effektiv die Teilergebnisse aus lokalen Berechnungen und vermeidet Konflikte mit Updates, die gleichzeitig auf anderen Maschinen berechnet werden. In jeder Berechnungs-Runde werden Schritte einer beliebigen dualen Optimierungsmethode auf den lokalen Daten jeder Maschine parallel durchgeführt. Zum Schluss wird

dann nur ein einzelner Aktualisierungsvektor an den Masterknoten übermittelt.
So ist es möglich bei beispielsweise H Iterationen einer Online-Optimierungsmethode lokal pro Runde, einen Faktor von H in Bezug auf die Kommunikation im Vergleich zu dem naiven verteilten Aktualisierungsschema einzusparen [JST+14].

Für die lokalen Berechnungen in den Iterationen setzt Flink den *Stochastic dual coordinate ascent (SDCA)* Algorithmus ein [Fli19]. Der SDCA Algorithmus [SSZ13] löst für die Iterationen das Minimierungsproblem lokal und wird von dem CoCoA Algorithmus zu einem finalen Gradientenzustand aggregiert. Dieser Gradientenzustand wird dann über alle Knoten repliziert, um in weiteren Schritten verwendet zu werden.

Die Kombination von einer SVM mit CoCoA übertrifft die Leistung des klassischen Support Vektor Machine (SVM)-Algorithmus in einer verteilten Umgebung [KPSA18].

4 Apache Spark

Apache Spark [Spa19] ist ein Open-Source Cluster Computing Framework, welches 2009 an der University of California, Berkeley, gegründet wurde und seit 2013 Teil der Apache Software Foundation ist. Ziel war es, Probleme zu lösen, die mit der Hadoop MapReduce Verarbeitungsarchitektur einhergehen. So unterstützt Spark Anwendungen mit Working-Sets und hat dennoch eine vergleichbare Skalierbarkeit und Fehlertoleranz wie MapReduce [ZCF+10].
Spark besitzt die Fähigkeit der In-Memory Verarbeitung, wodurch Daten und Zwischenergebnisse einzelner Operationen temporär gespeichert werden. Daraus resultiert speziell bei iterativen Algorithmen, die den Hauptbestandteil des maschinellen Lernens bilden, ein enormer Geschwindigkeitsunterschied [KZ18]. Desweiteren erweitert Spark das zu MapReduce ähnliche Programmierungsmodell um eine Abstraktion für die gemeinsame Nutzung von Daten, welche „Resilient Distributed Datasets“oder auch RDD genannt wird. Mit diesen Erweiterungen kann Spark eine Vielzahl von Verarbeitungs-Workloads erfassen, welche bisher seperate Engines für SQL, Streaming, Machine Learning oder Graphenverarbeitung benötigten. Diese Implementierungen verwenden die gleichen Optimierungen wie spezialisierte Engines und erreichen ähnliche Leistung, laufen aber als Bibliotheken über eine gemeinsame Anwendung, sodass sie einfach und effizient zu erstellen sind [ZXW+16].

Dieses Kapitel stellt zunächst die Systemarchitektur von Apache Spark vor (4.1). Folgend wird die Streaming Verarbeitung in Apache Spark (4.2.1) und dessen beiden APIs vorgestellt. Zum Schluss wird das maschinelle Lernen in Apache Spark (4.3), die Implementierung der SVM (4.3.1) sowie die neue Bibliothek des maschinellen Lernens, namens Spark ML (4.3.2), präsentiert.

4.1 Systemarchitektur

Das Apache Spark System besteht aus mehreren Hauptkomponenten.
Wie man in Abbildung 4.1 erkennen kann, besteht der Kern der Systemarchitektur aus der Spark Core API, welche in Scala geschrieben wurde, jedoch ebenso Java, Python und R Schnittstellen anbietet und somit ein breites Feld an Anbindungsmöglichkeiten zur Verfügung stellt. Dieser Spark-Core bietet Basis-Funktionalitäten wie Scheduling, Speicherverwaltung, Fehlertoleranz und Monitoring, sowie die Verteilung der Daten [KKWZ15]. Zudem umfasst der Spark-Core auch das Resilient Distributed Dataset, eine im Cluster partitionierte Datenstruktur [And15]. Hierauf wird in Abschnitt 4.1.1 näher eingegangen.

Desweiteren beinhaltet das System die übergeordneten Bibliotheken: Spark's MLlib für maschinelles Lernen, GraphX für die Graphenanalyse, Spark Streaming für die Streamverarbeitung und Spark SQL für die strukturierte Datenverarbeitung. Das System entwickelt sich schnell durch Änderungen an seinen Kern-APIs und dem Hinzufügen von Bibliotheken weiter [SDC+16].

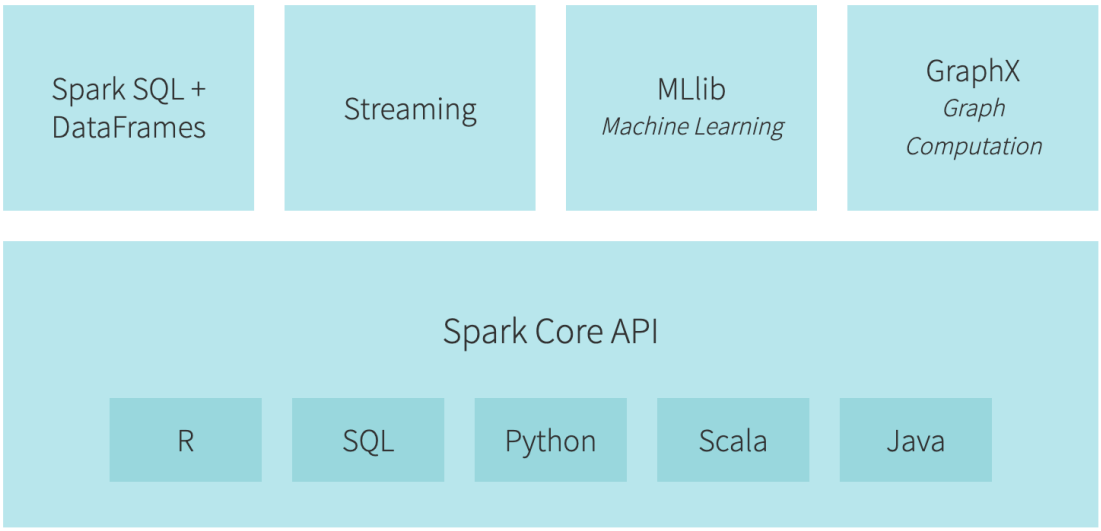

Abbildung 4.1: Spark Systemarchitektur:Komponenten *(Quelle: https://databricks.com)*

Spark kann verteilte Datensätze aus jeder Datei erstellen, die im Hadoop Distributed Filesystem oder einem anderen Speichersystem abgelegt ist, das von den Hadoop APIs unterstützt wird (bspw. Lokal, Amazon S3, Cassandra, Hive, etc.). Dabei benötigt Spark Hadoop selbst nicht. Es unterstützt lediglich Speichersysteme, die die Hadoop API im-

plementieren [KKWZ15]. Für die Verwaltung der gemeinsamen Resourcennutzung zwischen den Spark Anwendungen wird ein Cluster Manager verwendet. Dessen Aufgabe es ist, Cluster Resourcen für die Ausführung von Jobs zu ermitteln. Es wird also eine Master-Slave-Architektur verwendet, in der ein Master-Knoten Aufgaben an die Slaves (auch Worker genannt) verteilt, wodurch flexibel skaliert werden kann. Spark Core läuft über verschiedene Clustermanager wie Hadoop YARN, Apache Mesos, Amazon EC2 oder Spark's integriertem Clustermanager (Standalone) [SDC+16].

4.1.1 Resilient Distributed Dataset (RDD)

Formal ist ein Resilient Destributed Dataset eine schreibgeschütze Sammlung von Datensätzen, die nach bestimmten Regeln automatisch im Cluster partitioniert wird [XGFS13]. Ein RDD kann jede Art von Python, Java oder Scala Objekten sowie benutzerdefinierte Klassen enthalten [KKWZ15]. Sie können nur durch deterministische Operationen, auch Transformationen genannt, auf Daten im Dateisystem oder durch andere RDDs erstellt werden. Beispiele dieser Transformationen wären map, filter oder join.
Da RDDs nach der Erstellung schreibgeschützt sind, erzeugt jede Transformation auf ein RDD ein neues RDD [ZCD+12]. Ebenfalls müssen RDDs nicht immer materialisiert werden. Stattdessen besitzt ein RDD genug Informationen darüber, wie es von anderen Datensätzen abgeleitet wurde, um seine Partitionen aus Daten im Dateisystem zu berechnen. Daraus leitet sich ab, dass ein Programm nicht auf eine RDD verweisen kann, die es nach einem Fehler nicht rekonstruieren kann [ZCD+12]. Diese Strategie der Wiederherstellung erzielt eine hohe Toleranz im Cluster, ohne kostspielige Replikationen zu erzeugen.

Auf RDDs sind mehrere parallele Operationen ausführbar. Dabei unterscheidet man zwischen den oben genannten Transformationen und Aktionen. Transformationen sind lazy. Das bedeutet, dass Transformationen nicht sofort bei Aufruf ausgeführt werden, sondern erst, wenn das RDD erstmals durch eine Aktion verwendet wird. Diese Aktionen aggregieren ein RDD zu einem Wert, welcher dem Benutzer zur weiteren Verwendung zur Verfügung steht oder persistent für den späteren Gebrauch gespeichtert wird [And15]. Der Benutzer kann also bestimmen, welche RDDs er wiederverwenden möchte und eine Speicherstrategie (z.B. In-Memory) wählen. Ebenso kann er entscheiden, wie die RDDs partitioniert werden. Dabei basiert die Aufteilung auf einem Schlüssel, welcher in jedem RDD enthalten ist. Dies ist besonders nützlich für die optimale Platzierung. So kann

man beispielsweise sicherstellen, dass zwei Datensätze, die später miteinander verbunden werden sollen, auf die gleiche Weise hash-partitioniert sind [ZCD+12].

4.2 Streaming Verarbeitung in Spark

Für die Verarbeitung von Datenströmen besitzt Spark zwei eigene APIs namens *Spark Streaming* und *Structured Streaming*. In diesem Kapitel werden beide APIs vorgestellt und es wird auf die unterschiedlichen Konzepte eingegangen.

4.2.1 Spark Streaming

Die Spark Streaming API ermöglicht eine skalierbare, leistungsstarke und fehlertolerante Stream Verarbeitung von Real-Time Datenströmen. Die Daten können von verschiedenen Quellen wie Kafka, Flume, Kinesis oder TCP-Sockets aufgenommen und mit Hilfe komplexer Algorithmen verarbeitet werden. Diese Algorithmen werden durch High-Level Funktionen wie beispielsweise map, reduce oder window ausgedrückt. Abschließend können die verarbeiteten Daten in Dateisysteme, Datenbanken und Live-Dashboards ausgelagert werden [Spa19].
Da die Spark Engine jedoch auf Batch-Processing basiert, wird der potentiell unendliche Datenstrom durch die Spark Streaming API in Batches aufgeteilt. Der Streaming Dataflow in Abbildung 4.2 zeigt dies. Spark Streaming bekommt als Input einen wirklichen Stream, welcher in Batches zerlegt und an die Spark Engine weitergereicht wird. So ermöglicht es Spark, sein volles Spektrum an Bibliotheken der Graphenverarbeitung oder des maschinellen Lernens auch für die Stream Verarbeitung bereitzustellen.

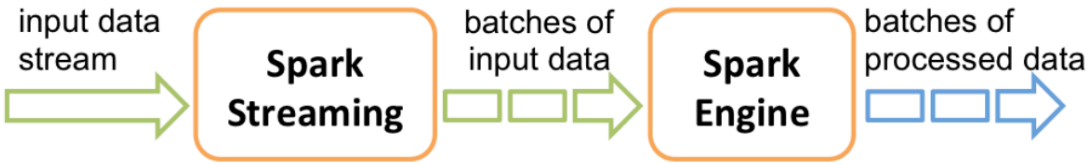

Abbildung 4.2: Spark Streaming Dataflow *(Quelle: https://spark.apache.org)*

Ein grundlegender Aspekt bei der Arbeit mit Spark sind die in Kapitel 4.1.1 erklärten RDDs. Um auch in der Streaming Welt mit diesen zu agieren, besitzt Spark Streaming eine Abstraktion namens Discretized Streams (D-Streams). Die Idee hinter diesem Modell

ist es, Stream Berechnungen als eine Reihe von deterministischen Batch Berechnungen in kleinen Zeitabständen zu behandeln. Die in jedem Intervall empfangenen Eingangsdaten werden über den Cluster verteilt gespeichert. So entsteht ein Datensatz für jedes Intervall. Nach Ablauf des Intervalls wird dieser Datensatz durch parallele Operationen wie map, reduce oder groupBy verarbeitet, um neue Datensätze zu erzeugen, aus denen Programmausgaben oder Zwischenstände hervorgehen. Diese Ergebnisse werden in RDDs gespeichert.

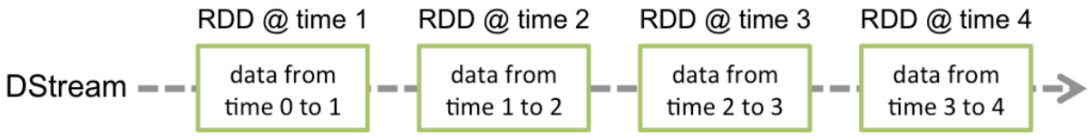

Abbildung 4.3: Spark D-Stream *(Quelle: https://spark.apache.org)*

Wie die Abbildung 4.3 zeigt, vereint ein D-Stream also eine Reihe von RDDs. So wird es dem Client ermöglicht, diese durch Operatoren zu manipulieren [ZDL+12].
Da D-Streams dem gleichen Verarbeitungsprozess wie Batch Systeme folgen, können diese beiden problemlos kombiniert werden. So kann man D-Streams mit statischen RDDs, wie einer geladenen Datei kombinieren, um beispielsweise eintreffende Nachrichten mit einem vorberechneten Spamfilter zu filtern oder den D-Stream mit historischen Daten zu vergleichen [ZDL+12].

4.2.2 Structured Streaming

Ebenso wie Spark Streaming arbeitet Structured Streaming mit eingehenden Datenströmen, welche als *Micro-Batches*, also sehr kleinen Datensätzen, weiterverarbeitet werden. Im Gegensatz zu Spark Streaming basiert Structured Streaming jedoch auf der Spark SQL Engine, welche durch das Arbeiten mit der *Dataset API* und der *DataFrame API* verwendet werden kann, um Streaming-Aggregationen, Ereigniszeitfenster oder Datenstrom-zu-Batch Joins auszudrücken. DataFrames sind Datensätze, welche in benannten Spalten organisiert sind. Konzeptionell sind diese äquivalent zu einer Tabelle in einer relationalen Datenbank. DataFrames ermöglichen eine strukturelle Datenverarbeitung, wodurch die Datenreinigung und Datenvorverarbeitung vereinfacht werden kann [Spa19]. Datasets hingegen bilden eine neue Schnittstelle, die in Spark 1.6 hinzugefügt wurde und die Vorteile von RDDs (starke Typisierung, Möglichkeit der Verwendung

leistungsstarker Lambda-Funktionen) mit den Vorteilen der optimierten Ausführungsmaschine von Spark SQL verbindet. Structured Streaming bildet also die Möglichkeit, *SQL Abfragen* auf Datenströmen aufzurufen.

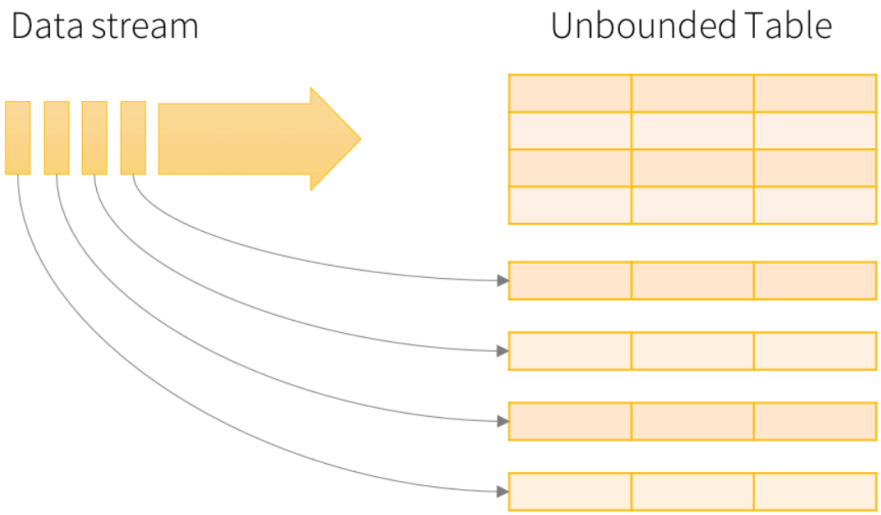

Abbildung 4.4: Structured Streaming Konzept *(Quelle: https://spark.apache.org)*

Abbildung 4.4 zeigt die Verarbeitung des Datenstromes. Jedes Datenelement, das auf dem Stream ankommt, ist wie eine neue Zeile, die an die Eingabetabelle angehängt wird. Eine Abfrage der Eingabe erzeugt eine Ergebnistabelle. Jedes Intervall werden neue Zeilen an die Eingabetabelle angehängt, wodurch schließlich die Ergebnistabelle wieder aktualisiert wird. Dabei ist zu beachten, dass Structured Streaming nicht die gesamte Tabelle materialisiert. Es werden nur die neusten verfügbaren Daten aus der Streaming-Quelle gelesen, schrittweise verarbeitet und die Ergebnistabelle aktualisiert. Danach werden die Quelldaten wieder verworfen. Lediglich minimale Zwischenzustände, welche zur Aktualisierung der Ergebnistabelle benötigt werden, bleiben erhalten. Für die Ausgabe der Ergebnisse bietet Structured Streaming verschiedene Sinks an. Durch die einfache Eingabe des Ausgabeformats, sei es Kafka, Memory oder die Konsole, wird festgelegt, wohin die Daten geschrieben werden.

4.3 Machine Learning in Spark

MLlib ist Sparks Bibliothek des maschinellen Lernens. Ziel ist es das praktische maschinelle Lernen einfach und skalierbar zu gestalten. Deshalb ist MLlibs Design und Philosophie einfach erklärt: Es ermöglicht den Aufruf verschiedener Algorithmen für verteilte Datensätze, wobei alle Daten als RDDs dargestellt werden. Es ist also eine einfache Menge von Funktionen, die auf RDDs aufgerufen werden [KKWZ15]. Dazu nutzt MLlib das reichhaltige Ökosystem von Spark, um eine High-Level-API anzubieten, welche mehrere Programmiersprachen unterstützt. Die angebotenen High-Level Funktionen beinhalten klassische Algorithmen des maschinellen Lernens wie *classification*, *regression* oder *clustering* [Spa19]. MLlib vereinheitlicht APIs des maschinellen Lernens, um es einfacher zu machen, mehrere Algorithmen in einer einzigen Pipeline oder einem Workflow zu kombinieren und so große Datenmengen optimal analysieren zu können.

Die enge Verbindung von MLlib zu Spark bietet dabei mehrere Vorteile. So hilft Sparks In-Memory Technologie und simultane Verarbeitung bei der Berechnung großer Datenmengen [KZ18]. Da Spark mit Blick auf iterative Berechnungen entwickelt wurde, ermöglicht es die Entwicklung effizienter Implementierungen von groß angelegten maschinellen Lernalgorithmen, welche typischerweise iterativ sind. Ebenso führen Änderungen der Low-Level Komponenten von Spark oft zu Leistungssteigerungen von MLlib, ohne direkte Änderungen an der Bibliothek selbst vorzunehmen [MBY+16].

4.3.1 Support Vector Machine Implementierung in Spark

Genauso wie Apache Flink besitzt Spark eine eigene SVM Implementation. Diese Implementation adapiert für das Minimierungsproblem einen verteilten *Stochastic Gradient Descent (SGD)* Algorithmus, welcher an den *Batch Gradient Descent* Algorithmus angelehnt ist. Der Batch Gradient Descent Algorithmus berechnet für jede Iteration des Gradientenabstiegs alle Trainingsbeispiele. Dies macht ihn bei einer großen Anzahl an Trainingsdaten rechnerisch sehr teuer. Der SGD Algorithmus löst dieses Problem, indem er die Berechnung drastisch vereinfacht. Anstatt den Gradienten genau zu berechnen, schätzt jede Iteration diesen Gradienten auf der Grundlage eines einzelnen zufällig ausgewählten Beispiels [Bot12]. In diesem Fall werden Mini-Batches als Stichprobe verwendet, um Teilgradienten in jeder Iterations-Phase zu berechnen. Für die Aktualisierung des globalen Verlaufs werden dann nur diese Teilergebnisse über das Netzwerk versendet.

4.3.2 Spark ML

Ein wichtiger Bestandteil des maschinellen Lernes ist das Einbinden einer *Lernpipeline*. Diese beinhaltet oft eine Abfolge von Datenvorverarbeitungen, Merkmalsextraktionen, Modellanpassungen und Validierungsphasen. Dieses Paket, genannt Spark.ml, vereinfacht die Entwicklung und Abstimmung von mehrstufigen Lernpipelines, indem es einen einheitlichen Bestand von High-Level-APIs bereitstellt [MBY+16].
Grundlegend für Spark.ml sind *DataFrames*, welche auf der Spark SQL API basieren (Siehe 4.2.2). Eine Spark.ml Pipeline besteht aus einem oder mehreren *Transformern* und einem oder mehreren *Estimators*. Transformer erhalten ein DataFrame als Eingabe und überführen dies in ein neues DataFrame, meist mit einer neuen Spalte. Estimators erhalten ebenso ein DataFrame, erzeugen jedoch ein Modell, welches wieder als Transformer in der Pipeline dient. So kann z.B. ein einfacher Arbeitsablauf zur Verarbeitung von Textdokumenten mehrere Phasen umfassen:

1. Teilen des Textes jedes Dokuments in Wörter.
2. Konvertieren der Worte jedes Dokuments in einen numerischen Merkmalsvektor.
3. Erlernen eines Vorhersagemodells unter Verwendung der Merkmalsvektoren

Die ersten beiden Phasen sind in diesem Fall Transformer, Phase 3 ein Estimator. Die gesamte Pipeline selbst ist also ein Estimator, da diese nach Abschluss der Transformer und Estimator ein Pipeline Modell erzeugt, welches als Transformer für kommende Daten dienen kann.
In Abbildung 4.5 werden die oben genannten drei Schritte für die Verarbeitung von Textdokumenten dargestellt. Die ersten beiden Schritte sind Transformer (*Tokenizer* und *HashingTF*). Sie teilen das Dokument in Wörter auf und erstellen den numerischen Merkmalsvektor. *Logistic Regression* ist der Estimator, der ein Logistic Regression Modell erzeugen wird, welches später als Klassifikator dienen kann.

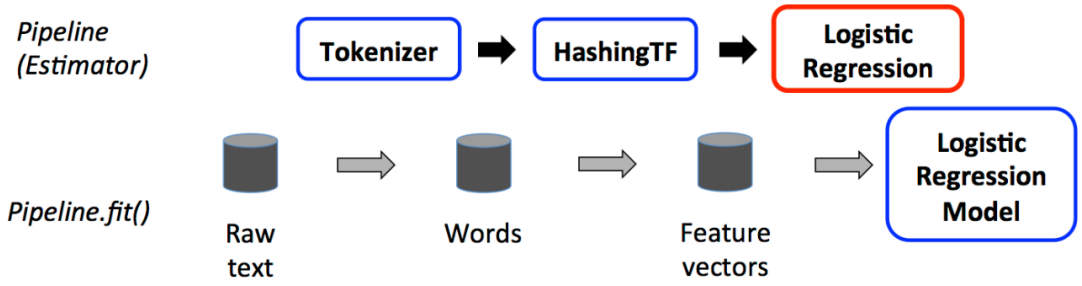

Abbildung 4.5: Spark Pipeline als Estimator *(Quelle: https://spark.apache.org)*

Nach dem Erstellen des Modells dient die gesamte Pipeline nun als Transformer, da kein weiteres Modell mehr erzeugt wird, sondern eintreffende DataFrames durch die Pipeline laufen und transformiert werden. Die Ausgabe der Pipeline ist also ein veränderes DataFrame. In Abbildung 4.6 gibt es nun keinen Estimator mehr. Der letzte Schritt wurde durch das erzeugte Logistic Regression Modell ersetzt.

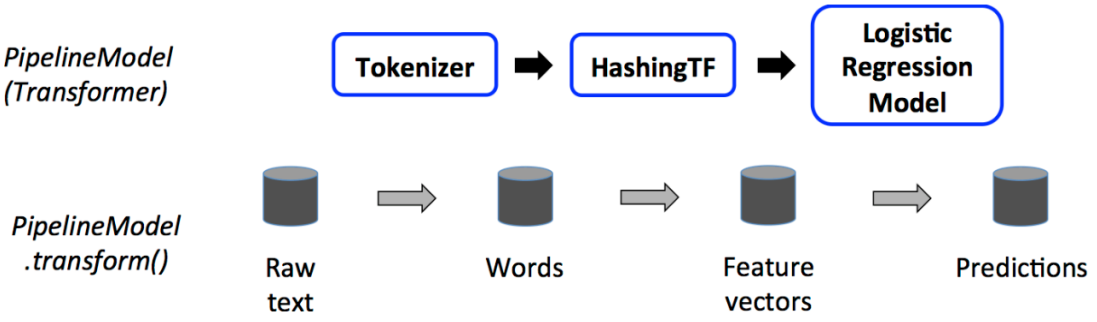

Abbildung 4.6: Spark Pipeline als Transformer *(Quelle: https://spark.apache.org)*

5 Related Work

Zu dem in dieser Arbeit enthaltenen Thema Flink vs. Spark gibt es einige wissenschaftliche Artikel. Dieses Kapitel stellt die schon herausgearbeiteten Erkenntnisse dieser Forschungen dar und soll als Zusammenfassung des momentanten Status im Bereich des oben genannten Themas dienen.

In dem Artikel *„A comparison on scalability for batch big data processing on Apache Spark and Apache Flink“* von Diego Garcia-Gil, wird die native Implementierung des SVM Algorithmus in Apache Flink und Apache Spark auf einem Cluster mit 9 Nodes und einem Master analysiert.
Für den Versuch wurde der *ECBDL14* Datensatz verwendet. Dieser Datensatz stammt aus dem ML-Wettbewerb der *Evolutionary Computation for Big Data and Big Learning*, welcher am 14. Juli 2014 im Rahmen der internationalen Konferenz GECCO-2014 stattfand. Er besteht aus 631 Merkmalen (einschließlich numerischer und kategorischer Attribute) und 32 Millionen Instanzen. Es handelt sich dabei um ein binäres Klassifizierungsproblem, bei dem die Klassenverteilung mit nur 2 Prozent positiven Instanzen stark unausgewogen ist. Deshalb wurde ein Algorithmus zur Vorverarbeitung angewendet, welcher die Dysbalance der beiden Klassen ausgleichen sollte. Der *Random OverSampling (ROS)* Algorithmus replizierte die Instanzen der Minderheitenklasse aus dem ursprünglichen Datensatz, bis die Anzahl der Instanzen der beiden Klassen ausgeglichen war. Daraus resultierte ein Datensatz mit 65 Millionen Instanzen.
Der ursprüngliche Datensatz wurde nach dem Zufallsprinzip mit fünf unterschiedlichen Anteilen gesampelt, um die Skalierbarkeitsleistung beider Frameworks zu messen. Für diesen Test wurden 10, 30, 50, 75 und 100 Prozent des vorverarbeiteten Datensatzes verwendet. Die SVMs wurden auf 100 Iterationen mit einer Schrittweite von 0,01 und einem Regularisierungsparameter von 0,01 festgelegt.

Die Ergebnisse dieses Tests sind in der Abbildung 5.1 dargestellt. Die erste Spalte zeigt den jeweils benutzen Datensatz, die folgenden Spalten geben die Lerndauer in Sekunden an. Wie man an der Abbildung erkennen kann, skaliert Spark wesentlich schneller als

Dataset	Spark MLlib	Flink	Difference
ECBDL14-10	42	111	69
ECBDL14-30	61	196	135
ECBDL14-50	103	302	199
ECBDL14-75	123	456	333
ECBDL14-100	174	783	609

Abbildung 5.1: Flink/Spark - SVM Lernzeit in Sekunden *(Quelle: [GGRGGH17])*

Flink. Der Zeitunterschied zwischen Spark und Flink nimmt mit der Größe des Datensatzes zu, Flink ist anfangs 2,5x langsamer, mit dem gesamten Datensatz sogar 4,5x. Die gesamte Analyse beinhaltete neben SVM auch den Lernalgorithmus *Lineare Regression*. Auch auf diesem Gebiet war Spark deutlich vor Flink. Deshalb lautet das Endresultat dieses Versuchsaufbaues, dass sich Apache Spark als das Framework mit besserer Skalierbarkeit und insgesamt schnelleren Laufzeiten erwiesen hat [GGRGGH17].

[CDE+16] Ein weiterer Vergleich, welcher mehr den Fokus auf die Stream-Verarbeitung legte, fand in dem Artikel *„Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming“* von Sanket Chintapalli statt und wurde 2016 auf dem *IEEE International Parallel and Distributed Processing Symposium Workshop* veröffentlicht.

In diesem Experiment wurde ein Leistungsvergleich der Frameworks Flink, Spark und *Apache Storm* in Bezug auf die 99. Perzentil-Latenz in Relation mit der Durchsatzmenge ausgeführt. Die Analyse des Frameworks Apache Storm sowie dessen Eigenschaften und Arbeitsweise werden hier bewusst nicht weiter erläutert, da es nichts mit dem in dieser Thesis durchgeführten Versuch zu tun hat.

Der Versuchsaufbau simulierte eine Pipeline für Werbeanalysen. Diese Pipeline enthielt eine Reihe von Werbekampagnen und eine Reihe von Anzeigen für jede Kampagne. Die Aufgabe des Tests bestand darin, verschiedene JSON-Ereignisse von *Apache Kafka* [Kaf19] zu lesen, die relevanten Ereignisse zu identifizieren und eine fensterbasierte Anzahl von relevanten Ereignissen pro Kampagne in *Redis* [Red19] zu speichern. Der verwendete Cluster verfügt über insgesamt 40 Nodes, von denen bis zu 30 in diesem Test verwendet wurden. Für alle drei Frameworks wurde die Plattform mit 10 Arbeiterknoten und einem Koordinationsknoten konfiguriert.
Das Ergebnis dieses Experiments ist in Abbildung 5.2 zu sehen. Die X-Achse beschreibt die Durchsatzrate der Events pro Sekunde, die Y-Achse die 99. Perzentil-Latenz in Millisekunden.

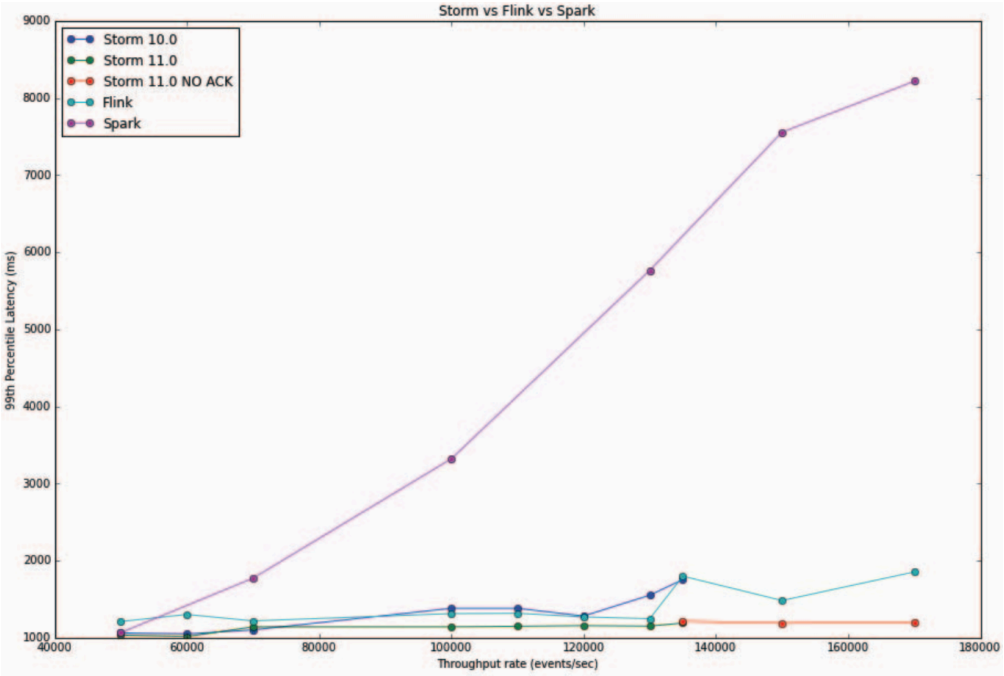

Abbildung 5.2: Flink/Spark/Storm - Throughput *(Quelle: [CDE+16])*

Wie deutlich zu erkennen ist, hat Spark eine viel höhere Latenz als Flink. Flink verhält sich wie ein echtes Streaming-Verarbeitungssystem, dessen Latenz ziemlich linear verläuft, da es versucht Events direkt zu verarbeiten, sobald diese verfügbar sind. Es ist aber zu erwarten, dass Spark durch Konfigurieren eines höheren Batch-Intervalls mehr Durchsatz erzielen kann [CDE+16].

6 Hate-Speech Filter Algorithmus

Dieses Kapitel soll Aufschluss über die Umsetzungen des Hate-Speech Filters in den einzelnen Anwendungen geben, um nachvollziehen zu können, was die Stärken der Frameworks in diesem Vorhaben ausmachen, aber auch mit welchen Kompromissen oder Komplikationen die Anwendungen implementiert wurden.

Zunächst wird die geplante Grundstruktur der Anwendung (6.1) dargestellt. Folgend werden die unterschiedlichen Implementationen von Apache Flink (6.2) und Apache Spark (6.3) vorgestellt.

6.1 Grundstruktur

Die Grundstruktur und das Vorgehen der beiden Applikationen ist dabei identisch: Da eine SVM nicht mit Wörtern und Sätzen arbeiten kann, muss eine *Pre-Processing-Pipeline* definiert und geschrieben werden, welche einen Tweet zu einem Vektor verarbeitet. Diese Pipeline beinhaltet das Säubern des Tweets von irrelevanten Daten wie Links zu anderen Webseiten oder Erwähnung anderer Nutzer im Tweet. Ebenso werden sogenannte *Stop Words* entfernt. Stop Words bezeichnen häufig verwendete Wörter wie zum Beispiel „*the*“, „*a*“, „*an*“ oder „*in*“. Diese Wörter sind für die Klassifizierung eines Tweets nicht von Bedeutung, da sie in so gut wie jedem Tweet vorkommen und nichts über den Kerngedanken eines Tweets aussagen.
Damit ein Wort in seinen verschiedenen Formen, wie beispielsweise Singular oder Plural, dennoch als gleiches Wort erkannt wird, benötigt man zudem einen *Stemmer*. Die Aufgabe eines Stemmers ist es, Wörter zu ihrem gemeinsamen Wortstamm zurückzuführen, wobei jeder Stemmer Algorithmus dies in einer anderen Art und Weise umsetzt. Für die beiden Implementationen wurde der *Porter-Stemmer* Algorithmus von Smile [Li19] verwendet. Dieser Algorithmus beinhaltet verschiedene Verkürzungsregeln für Suffixe. So werden die Wortendung „*ies*“ und „*y*“ auf die Wortendung „*i*“ gekürzt.

Dies hat zur Folge, dass die Wörter „*Library*“ und „*Libraries*“ unter dem gemeinsamen Wortstamm „*Librari*“ als gleiches Wort erkannt werden.
Der letzte Schritt der Pipeline ist das Vektorisieren des gereingten Tweets.
Dafür wird in beiden Anwendungen ein *TF-IDF* Algorithmus verwendet, welcher die Häufigkeit eines Wortes in einem Text mit der Häufigkeit des Vorkommens dieses Wortes in den anderen Texten in Relation setzt. Die hier genannten anderen Texte bilden dabei den *Corpus* (dt. Textkorpus). Dieser muss abgespeichert werden, um im späteren Verlauf für die einzelnen Wörter den TF-IDF Wert zu berechnen.
Durch die Wahl dieses Algorithmus wird Wörtern, die sehr häufig in anderen Texten vorkommen, ein niedrigerer Wert zugeschrieben, da das Vorkommen seltener Begriffe für die Relevanz aussagekräftiger ist als das Vorkommen häufig verwendeter Begriffe, wie beispielsweise die erwähnten Stop Words. Das Resultat der Pipeline ist also ein vektorisierter Tweet, bei dem jedes Wort durch einen TF-IDF Wert ersetzt wurde. Das Erstellen des Pre-Processing-Pipeline Modells ist in der Abbildung 6.1 veranschaulicht.

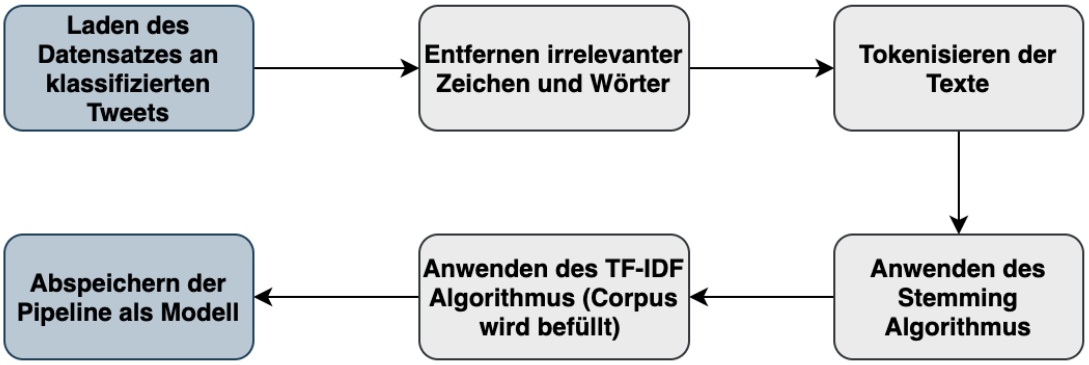

Abbildung 6.1: Erstellung einer Pre-Processing-Pipeline

Als nächstes wird eine SVM mittels bereits klassifizierter Daten, welche die oben beschriebene Pipeline durchliefen, trainiert, um im späteren Verlauf auf diesen Daten basierende Entscheidungen zu treffen. Die Daten werden dafür in Trainings- und Testdaten unterteilt. Zuerst erlernt die SVM die klassifizierten Trainingsdaten, danach wird die Fehlerquote und Genauigkeit der SVM anhand der Testdaten überprüft. Die Ergebnisse der Klassifizierung der Testdaten dienen dann zur Optimierung der SVM mittels gegebener Parameter. So kann beispielsweise der Schwellenwert hoch oder runter gesetzt werden, ab wann ein Tweet als Hate-Speech klassifiziert wird oder die Anzahl der Iterationen eingestellt werden, die letztendlich entscheidet, wie sehr sich die Gewichtsvektoren der SVM an die Trainingsdaten anpassen. Dieses Modell der trainierten SVM muss darauf-

hin abgespeichert werden, da man sonst bei jedem Durchlauf dieses erneut trainieren müsste, was wieder Zeit in Anspruch nehmen und den Verlust einer bereits trainierten und optimierten SVM bedeuten würde. Abbildung 6.2 zeigt den Ablauf der Erstellung eines SVM Modells.

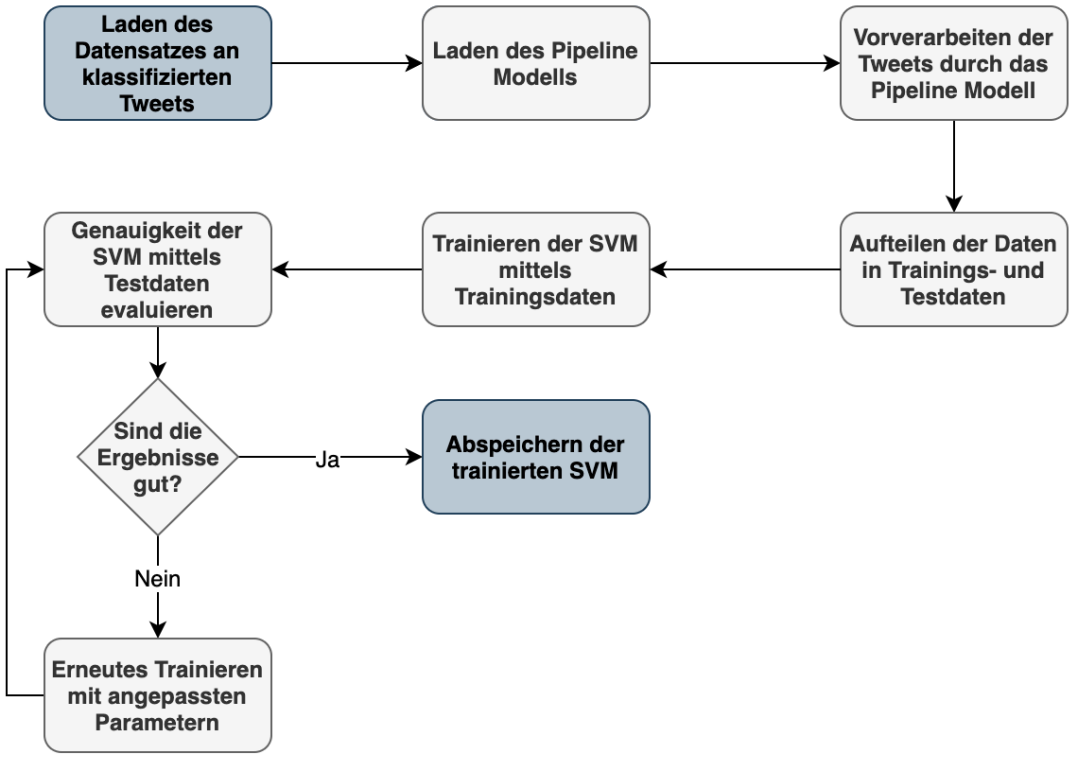

Abbildung 6.2: Training der SVM

Der in diesem Versuch benutzte Datensatz für den Corpus und die SVM stammt aus dem Artikel *„Automated Hate Speech Detection and the Problem of Offensive Language“* von Thomas Davidson [DWMW17] welcher 24784 klassifizierte Tweets beinhaltet.

Besitzt man Pipeline und SVM, muss eine Verbindung zur Twitter API hergestellt werden, von der aus Tweets über den Datenstrom zuerst nach ihrer Sprache gefiltert werden, durch die Pipeline laufen und danach von der SVM klassifiziert werden. Dieser Anwendungverlauf ist in der Abbildung 6.3 dargestellt.

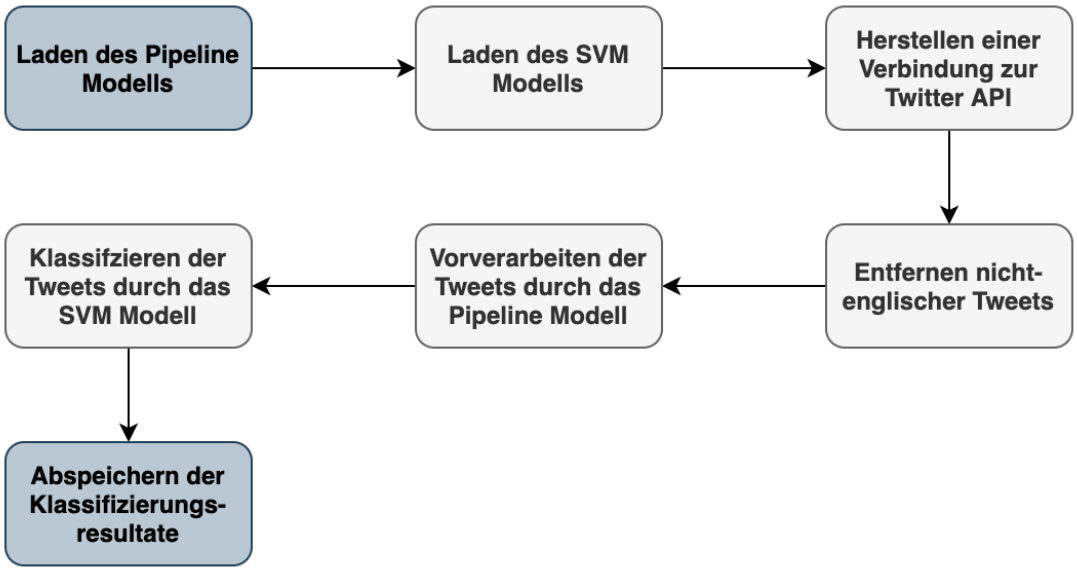

Abbildung 6.3: Real-Time-Twitter-Filter Anwendung

6.2 Apache Flink Implementierung

Für diese Anwendung wurde die Scala Version 2.12 und die zu diesem Zeitpunkt neueste Apache Flink Version 1.9.1 verwendet.
Apache Flink besitzt einen eigenen Twitter-Connector. Dieser benötigt lediglich Consumer-Keys und Access-Tokens, um die Verbindung zur Twitter API herzustellen. Daraus resultiert ein Datenstrom, welcher die in Kapitel 2.3 erklärten JSON-Objekten an die Applikation liefert.
Zum Erstellen der Pipeline bietet Flink wenig Hilfe an. Das Framework besitzt zwar die elementaren Algorithmen des maschinellen Lernens, bietet aber wenig Unterstützung im Bereich *Natural Language Processing (NLP)*, der maschinellen Verarbeitung natürlicher Sprache. Das Framework besitzt keinen eigenen TF-IDF Algorithmus, Stemmer, oder ein für diese Zwecke geeignetes Pipeline-Modell. Deshalb wurde der Porter-Stemmer von Smile [Li19] eingebunden. Der TF-IDF Algorithmus sowie die oben genannte Vorsäuberung des Tweets wurden selbst entworfen und implementiert.
Es ist jedoch nicht möglich, ein Pipeline-Modell abzuspeichern, um es für spätere Versuche zu laden und erneut zu verwenden. Aus diesem Grund benötigt die Flink Anwendung vor Start des Twitter-Streams einige Sekunden, um den Datensatz für den Corpus erneut einzulesen.

Hauptproblem der Umsetzung dieser Anwendung war jedoch Flinks eigene SVM Implementation. Da diese auf der DataSet API basiert, ist sie nicht in der Lage, einen Datenstrom zu verarbeiten.
Zur Lösung dieses Problems wurden Teile des Streamline Projektes [Ben] an meine Anwendung angepasst und integriert. Das Streamline Projekt wird durch das Forschungs- und Innovationsprogramm der europäischen Union Horizon 2020 finanziert und arbeitet an der einheitlichen Verarbeitung von Batch- und Streamdaten. Dadurch kann eine von Batchdaten trainierte SVM einen Datenstrom auffassen und dessen Inhalte klassifizieren. Man benötigt also beide Ausführungsumgebungen der Flink API, die Batch-Umgebung, um die SVM zu trainieren und die Stream-Umgebung, um eintreffende Datenströme zu klassifizieren. Flink bietet nicht direkt die Möglichkeit, ein trainiertes SVM Modell abzuspeichern. Jedoch können die Gewichtsvektoren dieses Modells ausgegeben und zu einem späteren Zeitpunkt einem SVM Modell zugewiesen werden. Dies passiert leider nur über den Umweg des Auslesens einer CSV-Datei, was wieder einige Sekunden zu Beginn der Ausführung in Anspruch nimmt.

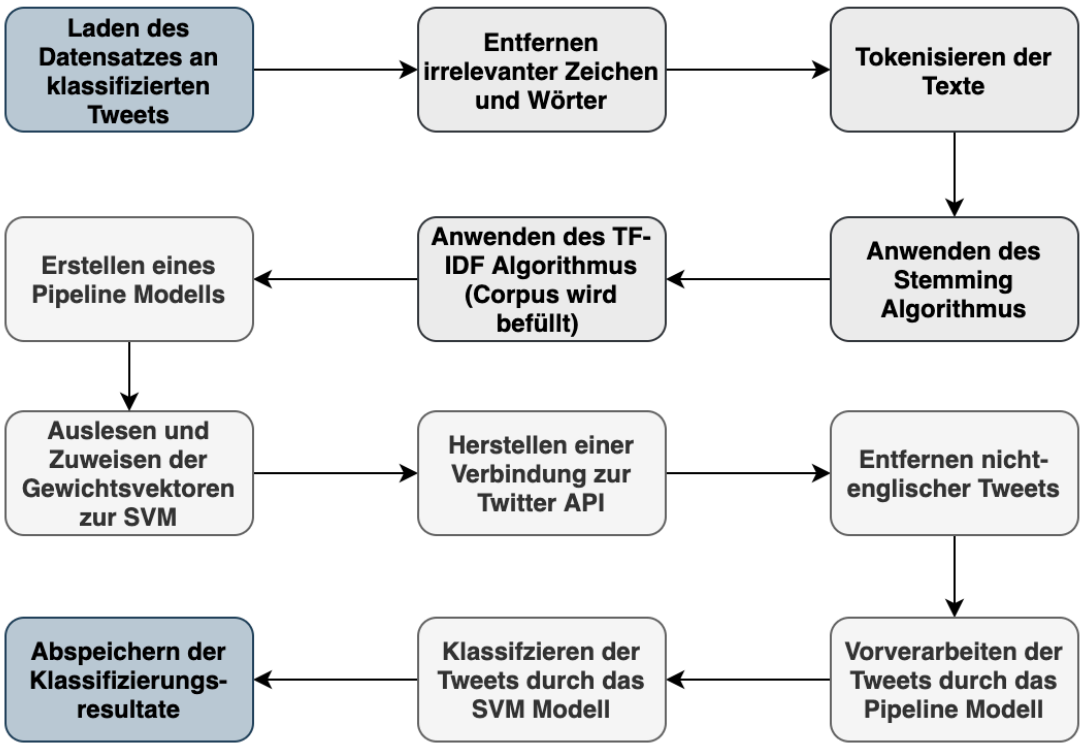

Abbildung 6.4: Umsetzung der Real-Time-Twitter-Filter Anwendung in Flink

Wie deutlich in der Abbildung 6.4 zu erkennen ist, benötigt die Flink Anwendung wesentlich mehr Zeit zu Beginn, da erst das Pipeline Modell erstellt wird und die SVM Gewichtsvektoren ausgelesen werden müssen. Der Programmverlauf der Hauptanwendung danach ist jedoch dem der Abbildung 6.3 entsprechend.

6.3 Apache Spark Implementierung

Für diese Anwendung wurde ebenfalls die Scala Version 2.12 sowie die neueste Apache Spark Version 2.4.4 verwendet.
Die Verbindung zu der Twitter API findet auch hier über einen Twitter-Connector statt, welcher die genannten Keys und Tokens zur Authentifizierung benötigt. Dieser Connector ist jedoch nicht selbst im Apache Spark Framework integriert, sondern erfolgt durch den Einbezug der Apache Bahir Bibliothek [Bah19]. Diese erweitert verteilte Analyseplattformen durch Streaming-Konnektoren und SQL-Datenquellen.

Das Pipeline Modell lässt sich bei Spark wesentlich einfacher gestalten.
Lediglich die Vorverarbeitung der Tweets in Form des Entfernens nicht relevanter Zeichen muss selbst implementiert und ein Stemmer integriert werden. Für den Stemmer wurde der gleiche Algorithmus von Smile [Li19] wie auch in der Apache Flink Anwendung verwendet. Spark bietet einen eigenen TF-IDF Algorithmus, Stop Word Entferner und die Möglichkeit, das definierte Pipeline Modell abzuspeichern. So kann die komplette Pipeline mit Corpus direkt zum Start kommender Anwendungen importiert und verwendet werden. Dank Spark ML kann sogar das SVM Modell in die Pipeline integriert werden. Dies ermöglicht im späteren Verlauf das einfache Aufrufen des Pipeline Modells, um sowohl die Vorverarbeitung als auch die Klassifizierung in einem Schritt durchzuführen.

Die Abbildung 6.5, welche den Programmverlauf der Spark Anwendung zeigt, verdeutlicht den Unterschied zur Umsetzung in Apache Flink. Bei Apache Spark ist der Programmablauf im Wesentlichen identisch zur Abbildung 6.3. Die Spark Anwendung ist sogar noch intuitiver durch das Einbinden der SVM in das Pipeline Modell.

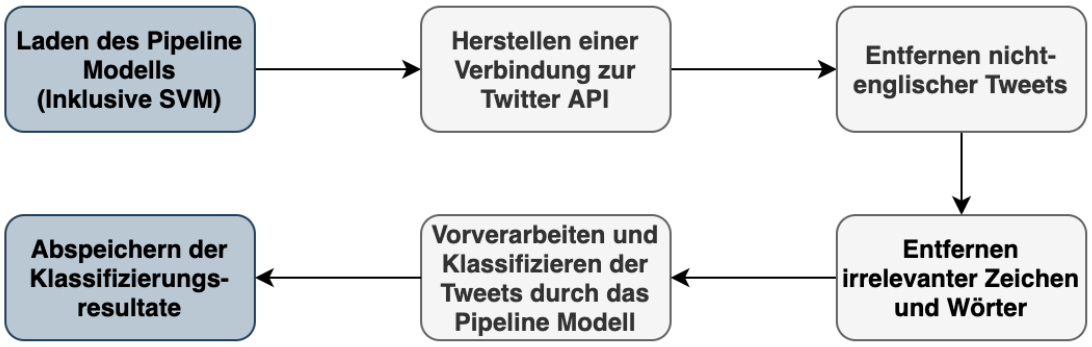

Abbildung 6.5: Umsetzung der Real-Time-Twitter-Filter Anwendung in Spark

7 Experimente

In diesem Kapitel wird die Leistung der Frameworks Apache Spark und Apache Flink anhand von Experimenten getestet. Dabei liegt der Fokus auf der Menge an Tweets, die eine Anwendung pro Sekunde verarbeiten kann, um zu testen, welches Framework mit hohen Lasten besser umgehen kann.
Zunächst wird im Kapitel 7.1 der Aufbau der Experimente erklärt. Darauf folgend werden die Ergebnisse im Kapitel 7.2 präsentiert. Zuletzt wird auf die Korrektheit der Experimente (7.3) und auf die gewonnenen Erkenntisse (7.4) eingegangen.

7.1 Aufbau der Experimente

Mit Hilfe der Experimente soll aufgezeigt werden, welche Stärken und Schwächen die beiden Frameworks im Bereich der Echtzeitverarbeitung von Daten haben. Dafür werden zunächst die Hypothesen aufgestellt, die sowohl allgemeine als auch spezielle Disziplinen abdecken sollen. Ebenso werden die Daten sowie das Cluster, auf dem die Experimente ausgeführt wurden, erläutert.

7.1.1 Hypothesen

Die folgenden Hypothesen dienen zum Vergleich der Performance der beiden Frameworks:

1. Die Menge an verarbeiteten Tweets pro Sekunde steigt proportional zu der Anzahl der Slaves im Cluster.
2. Apache Flink hat eine niedrigere Latenz als Apache Spark.
3. Die Verarbeitungszeit eines Tweets ist unter Apache Spark kürzer als unter Apache Flink.

4. Apache Flink hat einen geringeren Arbeitsspeicher-Verbrauch als Apache Spark.

Der Begriff Slave steht in der Hypothese 1 und im Folgenden dieser Thesis für die Taskmanager/Worker des jeweiligen Frameworks.

7.1.2 Begründung für die Wahl der Hypothesen

Beide Frameworks sind auf die Ausführung in verteilten Systemen spezialisiert. Es ist also die Annahme zu treffen, dass jeder weitere Slave/Taskmanager die Anwendung beschleunigt. Die Hypothese 1 nimmt deswegen an, dass die Menge an Tweets, welche eine Anwendung pro Sekunde klassifizieren kann, proportional zu der Anzahl der Slaves im Cluster steigt.
Sowohl Apache Spark als auch Apache Flink sind beides weitverbreitete Frameworks, welche in Firmen mit großen Datenmengen zum Einsatz kommen. Jedoch wurde Apache Spark von Beginn an auf Batch-Verarbeitung optimiert und auch dessen Streaming APIs sind als Mini- oder Micro-Batch-Verarbeitung zu sehen. Apache Flink hingegen bietet eine tatsächliche Streaming-Verarbeitung. Daraus ergibt sich die Hypothese 2, dass Apache Flink eine niedrigere Latenz als Apache Spark haben müsste. Auch wenn der Zugriff auf einen Tweet durch Apache Flink schneller erfolgen sollte, müsste die Verarbeitungszeit auf Grund der vorgefertigten Pipeline-Struktur von Apache Spark kürzer sein als unter Apache Flink. Daraus resultiert die Hypothese 3. Apache Flink verwaltet im Gegensatz zu Apache Spark seinen Speicher aktiv selbst, was vermuten lässt, dass der Arbeitsspeicherverbrauch geringer als bei Apache Spark ausfällt (Hypothese 4).

7.1.3 Cluster

Apache Spark sowie Apache Flink sind Cluster-Computing Frameworks. Um das Potenzial der beiden Frameworks voll auszuschöpfen, finden die Experimente auf einem Cluster mit einem Master und bis zu 3 Slaves statt. Alle fünf Maschinen besitzen 32 Gigabyte RAM, 8 Prozessorkerne (Quad-Core mit Hyper-Threading) und 2 Terabyte Festplattenspeicher. Ebenso verwenden alle Maschinen das Betriebssystem Ubuntu 16.04 LTS (64-Bit), auf dem ohne Virtualisierung Apache Flink in der Version 1.9.1 und Apache Spark in der Version 2.4.4 installiert wurde. Beide Versionen stellen zum Zeitpunkt des Erstellens dieser Arbeit die neueste stabile Veröffentlichung des jeweiligen Frameworks dar. Als verteiltes Dateisystem wurde das *Hadoop Distributed File System verwendet (HDFS)*. Dafür wurde *Hadoop* in der Version 2.10.0 auf jedem Rechner auf dem Cluster installiert und eingerichtet. Das Cluster befindet sich in einem eigenen Subnetz, wodurch äußere Einflüsse minimiert werden sollen. Die einzelnen Rechner sind via Gigabit-Lan miteinander verbunden.

Ein weiterer Rechner, welcher sich im Cluster befindet, dient als TwitterSource für die beiden Anwendungen, da die von Twitter selbst angebotene API die Tweets pro Sekunde bis auf wenige begrenzt. Für die selbst gebaute Twitter API Simulation wurde ein Kafka Server auf dem Rechner aufgesetzt, welcher durch ein Python-Skript vorher heruntergeladene Tweets, in der selben Form, wie es auch die Twitter API tun würde, an die Anwendungen sendet. Da die tatsächliche Nutzung dieser Anwendungen aber über die Twitter API erfolgen würde, wurde auch diese in den Grafiken 6.4 und 6.5 dargestellt. Lediglich für die folgenden Experimente dient Kafka als Quelle der Anwendungen.

Für das Monitoring der Systemressourcen wurden über den Service *Telegraf* die Metriken der einzelnen Maschinen ausgelesen und in der *InfluxDB* persistiert. Zur Darstellung der Metriken diente das Monitoring Programm *Grafana*.
Ebenso wurde zur Messung der Tweets pro Sekunde, welche die Anwendungen klassifizieren konnten, *PostgreSQL* verwendet. PostgreSQL ist ein objektrelationales Datenbankmanagementsystem, welches mit dem Kafka Server verbunden wurde, um eingehende und ausgehende Tweets zu erfassen. Die grafische Darstellung dieser erfolgte ebenso über Grafana.

7.1.4 Daten

Wie schon in Kapitel 6 erwähnt, stammt der bereits klassifizierte Datensatz an Tweets aus dem Artikel *„Automated Hate Speech Detection and the Problem of Offensive Language“* von Thomas Davidson [DWMW17] und enthält 24784 klassifizierte Tweets. Die Daten, welche von der Kafka TwitterSource versendet werden, kommen aus Archiven der Internetseite *archive.org*[Kah19]. Dort enthalten sind gesammelte Tweets aus dem Jahre 2018, welche über die TwitterAPI entnommen wurden. Die Tweets sind dementsprechend in der gleichen Form eines Json-Objektes vorhanden und lassen sich ideal zum Nachahmen der Twitter API verwenden.

7.2 Durchführung

Die Experimente wurden anhand der Hypothesen mit variierender Anzahl an Slaves durchgeführt. Der Cluster selbst wurde von den Frameworks im Standalone-Modus verwaltet. Für die gesamten Experimente wurden die Standardkonfigurationen der Frameworks nur minimal verändert, um sie den Resourcen im Cluster anzupassen. So wurde bei Apache Flink die JVM Heap Size auf 8 Gigabyte und bei Apache Spark die Executor Memory auf 8 Gigabyte, sowie die Executor Cores auf 8 gesetzt. Zudem wurde die Parallelität eines Jobs bei Apache Flink auf die Anzahl der zu dem Zeitpunkt verfügbaren Kerne gesetzt. Apache Spark verwaltet seine Resourcen selbstständig und dynamisch abhängig von der Rechenlast. Um eine Vergleichbarkeit der Ergebnisse zu erreichen, waren die erwähnten Anpassungen, bis auf die Parallelität bei Apache Flink, durchgehend identisch. Die Messungen wurden jeweils mit einer Slave Anzahl von 1-3 fünf mal durchgeführt und die Ergebnisse arithmetisch gemittelt. Die Grafiken und die Tabellen, welche die Ergebnisse enthalten, stellen diese gemittelten Werte dar.

7.2.1 Hypothese 1

„Die Menge an verarbeiteten Tweets pro Sekunde steigt proportional zu der Anzahl der Slaves im Cluster.“

Zum Testen der ersten Hypothese wurden die beiden Frameworks zunächst mit einem Master und einem Slave gestartet und von der Kafka TwitterSource mit Daten beliefert. Diese Versuche fanden nicht gleichzeitig statt, sondern erst für Apache Flink und

folgend für Apache Spark. Wurde das Maximum an Tweets, welche pro Sekunde von den Anwendungen verarbeitet werden konnten erreicht, wurde nach einer Verweilzeit das Maximum, Minimum und der arithmetische Mittelwert gebildet. Danach wurde ein weiterer Slave dem Cluster hinzugefügt und die Tests wiederholt bis alle 3 Slaves Teil der jeweiligen Anwendung waren. Durch die Kafka TwitterSource konnte eine maximal Anzahl von 8500 Tweets pro Sekunde generiert werden.

Anzahl der Slaves	**1**	**2**	**3**
Maximum	3818,4	7095,2	7377,4
Minimum	3504,6	6497,6	6888,8
Mittelwert	3673,2	6809,2	7175,4

Tabelle 7.1: Apache Flink: Anzahl der verarbeiteten Tweets pro Sekunde

Die Tabelle 7.1 beinhaltet die Messergebnisse von Apache Flink. Die dargestellten Zahlen zeigen die von Flink verarbeiteten Tweets pro Sekunde mit der in der Kopfzeile enthaltenen Anzahl von Slaves im Cluster.
Zu sehen ist ein deutlicher Anstieg der Messwerte, wenn man von einem Slave auf zwei Slaves im Cluster erhöht. So vermehren sich die verarbeiteten Tweets im Maximum, Minimum und im Mittelwert um rund 85 Prozent. Bei der Erhöhung von zwei auf drei Slaves im Cluster steigen die Messwerte nur noch um rund 5 Prozent minimal an.

Anzahl der Slaves	**1**	**2**	**3**
Maximum	3832	7300	7345
Minimum	3216,6	6699	7001
Mittelwert	3475,8	6987	7143,6

Tabelle 7.2: Apache Spark: Anzahl der verarbeiteten Tweets pro Sekunde

Die Messwerte von Apache Spark, welche in Tabelle 7.2 dargestellt sind, zeigen ähnliche Ergebnisse. Der Wechsel von einem zu zwei Slaves im Cluster lässt die verarbeiteten Tweets pro Sekunde um rund 101 Prozent stark ansteigen. Die Erhöhung auf drei Slaves hat jedoch nur noch sehr geringe Auswirkungen. Der Mittelwert steigt um lediglich 2,24 Prozent. Entgegen der Annahme, dass die Anzahl der verarbeiteten Tweets pro Sekunde proportional zu der Anzahl der Slaves im Cluster steigt, spiegeln die Messergebnisse dies nicht wieder. Da beide Frameworks auf die Verarbeitung im Cluster spezialisiert sind, gibt es einen enormen Anstieg der Messwerte, wenn die Frameworks die Möglichkeit bekommen, Rechenaufgaben und Lasten zwischen den einzelnen Slaves aufzuteilen und so in der Lage sind, die Last des Einzelnen zu verringern.

Auffällig ist die Ähnlichtkeit der Messergebnisse der beiden Frameworks. Abbildung 7.1 vergleicht die Mittelwerte von Apache Flink und Apache Spark im Bezug auf die Anzahl der Slaves im Cluster. Wie deutlich zu erkennen ist, existiert kein signifikanter Unterschied zwischen den Leistungen der beiden Frameworks. Ebenso unterscheiden sich die Minima und Maxima der Frameworks nur gering voneinander. Apache Spark erzielt marginal bessere Ergebnisse als Apache Flink, dies könnte an dem selbst implementierten TF-IDF Algorithmus in der Apache Flink Anwendung liegen.

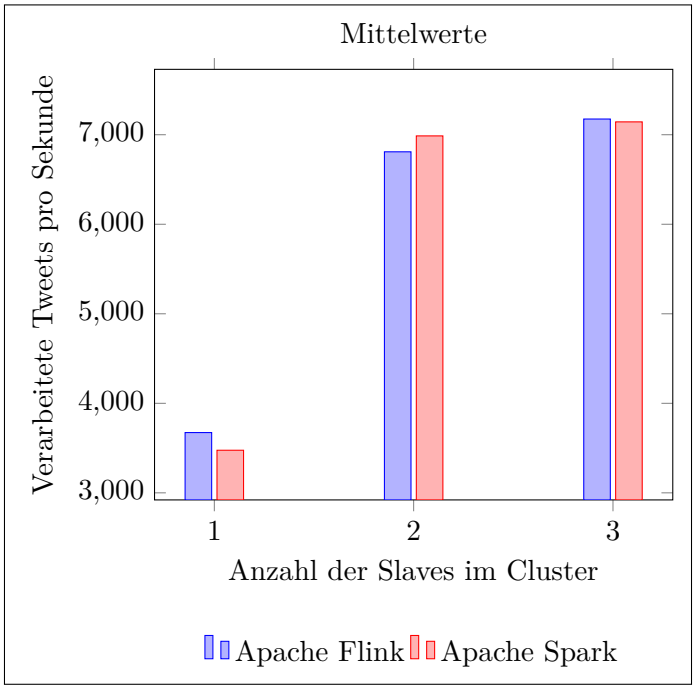

Abbildung 7.1: Vergleich der Mittelwerte im Bezug auf die Anzahl der Slaves im Cluster

7.2.2 Hypothese 2

„Apache Flink hat eine niedrigere Latenz als Apache Spark.“

Die Messwerte zum Überprüfen dieser Hypothese wurden mit Hilfe von zwei Zeitstempeln kalkuliert. Der erste Zeitstempel wurde bei der Erstellung des Tweets generiert. Der zweite Zeitstempel wurde beim Auffassen des Tweets von der jeweiligen Anwendung generiert.

Um die reinen Latenzen der Anwendungen zu messen, wurde zunächst die Grundlatenz des Sendens und Empfangens einer Nachricht von Kafka berechnet. Dafür wurde eine Nachricht, welche nur den Erstellungs-Zeitstempel enthielt, an Kafka gesendet und gemessen, wie schnell diese bei den Anwendung ankam. Diese Grundlatenz wurde von den späteren Messergebnissen abgezogen. Dies ist wichtig, da die Latenz, welche durch das Senden an Kafka entsteht, keinen Einfluss auf die Messwerte der jeweiligen Anwendung haben sollte. Interessant ist nur die pure Latenz, welche durch die Anwendungen selbst entsteht.

Anzahl der Slaves	**1**	**2**	**3**
Apache Flink	6,223s	5,642s	2,834s
Apache Spark	11,396s	9,967s	8,483s

Tabelle 7.3: Durchschnittswerte der Latenz in Sekunden bei 6000 Tweets pro Sekunde

Die Tabelle 7.3 zeigt zunächst den Einfluss der Anzahl der Slaves im Cluster auf die Latenz. Es ist deutlich zu erkennen, dass die Anzahl der Slaves im Cluster extrem relevant für die Latenz ist. Besonders bei Apache Flink liegen große Differenzen zwischen den einzelnen Werten, aber bei Apache Spark verbessern sich die Werte. Die Tabelle zeigt jedoch auch den enormen Unterschied der Latenzen der beiden Anwendungen. Die Latenz von Apache Flink ist in allen Fällen entschieden besser.

Abbildung 7.2 veranschaulicht diese Erkenntnis noch einmal. Dargestellt ist der Verlauf der Latenz mit drei Slaves im Cluster, bei steigender Azahl an eintreffenden Tweets. Von Beginn an liegt ein großer Unterschied zwischen den Latenzen der beiden Anwendungen. Auch im späteren Verlauf bleiben die Werte der Latenzen weiterhin weit voneinander entfernt. Diese Hypothese ist damit eindeutig bestätigt.

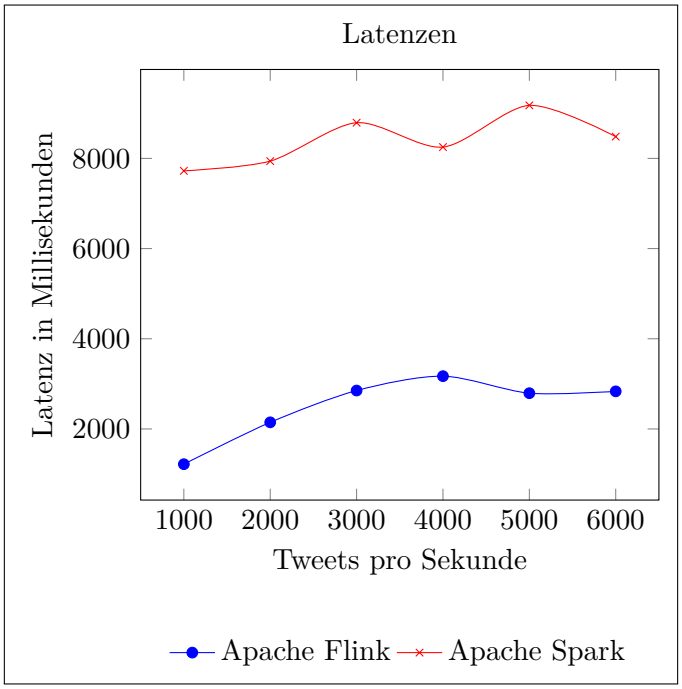

Abbildung 7.2: Mittelwerte der Latenzen mit drei Slaves im Cluster bei steigender Anzahl an eintreffenden Tweets

7.2.3 Hypothese 3

„Die Verarbeitungszeit eines Tweets ist unter Apache Spark kürzer als unter Apache Flink.“

Für diese Hypothese wurde der Zeitstempel des eintreffenden Tweets von dem Zeitstempel des verarbeiteten Tweets subtrahiert. Wie in Hypothese 2 wurde dieser Versuch mit variierender Anzahl an Slaves im Cluster und eintreffenden Tweets durchgeführt und folgend das arithmetische Mittel berechnet. Die genannten Variationen hatten dabei nur wenig Einfluss auf die Länge der Verarbeitungszeit.

	Apache Flink	**Apache Spark**
Maximum	0,634384ms	0,45560ms
Minimum	0,324933ms	0,25453ms
Mittelwert	0,383203ms	0,30946ms

Tabelle 7.4: Durchschnittswerte der Verarbeitungszeit in Millisekunden

Wie in Tabelle 7.4, welche die Maxima, Minima und Mittelwerte der Verarbeitungszeiten darstellt, zu erkennen ist, liegen große Unterschiede zwischen den Verarbeitungszeiten. Die Verarbeitungszeit unter Apache Spark ist rund 20 Prozent schneller als unter Apache Flink. Die Hypothese konnte damit bestätigt werden. Die absoluten Zahlenunterschiede sind dennoch so gering, dass diese wenig Einfluss auf die Leistung der Anwendung haben werden.

7.2.4 Hypothese 4

„Apache Flink hat einen geringeren Arbeitsspeicher-Verbrauch als Apache Spark."

Ebenso wie in Hypothese 1, wurden die Messwerte für dieses Experiment durch je fünfmaliges Ausführen der Anwendungen mit variierender Anzahl an verfügbaren Slaves durchgeführt. Die Höhe der von der Kafka TwitterSource gesendeten Daten waren dabei immer nur gering höher als die von den Anwendungen machbaren Verarbeitungen an Tweets pro Sekunde. Die Berechnungen des Arbeitsspeicherverbauchs der einzelnen Slaves wurden danach ebenfalls arithmetisch gemittelt.

Anzahl der Slaves	**1**	**2**	**3**
Apache Flink	4,736gb	3,7532gb	3,5932gb
Apache Spark	6,948gb	5,53gb	5,393gb

Tabelle 7.5: Vergleich: Durchschnittlicher Arbeitsspeicherverbrauch in Gigabyte im Bezug auf die Anzahl der Slaves im Cluster

Die Tabelle 7.5 zeigt die Messwerte dieses Experimentes. Der Unterschied des Arbeitsspeicherverbauchs lässt sich anhand der Tabelle deutlich erkennen. Apache Flink hat im Vergleich zu Apache Spark durchgehend einen wesentich geringeren Arbeitsspeicherverbrauch. Auffällig ist der Unterschied des Abfallens des Arbeitsspeicherverbrauchs beim Hinzufügen eines weiteren Slaves zum Cluster. Beide Frameworks konnten durch das Hinzufügen eines zweiten Slaves den Arbeitsspeicherverbrauch um rund 20 Prozent reduzieren und somit 1 Gigayte an Speicher einsparen. Nach dem Hinzufügen des dritten Slaves sinkt der Verbrauch jedoch bei Apache Flink nur noch um 4,26 Prozent und bei Apache Spark um 2,48 Prozent.

Die Annahme, dass Apache Flink einen geringeren Arbeitsspeicherverbrauch als Apache Spark besitzt, ist somit bestätigt.

7.3 Korrektheit der Experimente

Die Zeitmessungen, welche für die Latenzunterschiede und die Laufzeitmessungen vollzogen wurden, wurden anhand der Systemuhren erfasst. Die verarbeiteten Tweets pro Sekunde, sowie die Laufzeit der Anwendung sind bei Apache Flink mit Vorsicht zu betrachten, da der selbst implementierte TF-IDF Algorithmus einen wesentlichen Unterschied der Laufzeit bewirken könnte. Dieser wurde am Beispiel des TF-IDF Algorithmus von SMILE [Li19] implementiert und optimiert. Ebenso könnte die Adaption der Flink SVM des Streamline Projektes [Ben] Auswirkungen auf die Laufzeit und auf die Menge an verarbeitbaren Tweets pro Sekunde haben. Dennoch wurde bei beiden Frameworks versucht, möglichst die vom jeweiligen Framework bereitgestellten Implementationen zu verwenden. Auch die Schwankungen der Latenz der Kafka TwitterSource könnten Einfluss auf die Messergebnisse haben. Es wurde versucht, diesen Einfluss durch das Berechnen eines Mittelwertes der Grundlatenz zu verringern.

7.4 Auswertung der Experimente

Die Experimente zeigen, dass Apache Flink deutlich geringere Latenzen als Apache Spark vorweist. Erklärt wird dies durch den Versuch von Apache Flink, Events direkt bei der Erstellung zu verwerten, sobald diese verfügbar sind. Das Auf- und Absteigen der Latenzen der beiden Frameworks kann durch vereinzelntes Auftreten von Latenzspitzen auf Grund der TwitterSource begründet werden. Zwar wurde der Mittelwert dieser Latenzen als Grundlatenz von den Messwerten subtrahiert, trotzdem könnten bei den Schwankungen mehr oder weniger Spitzen aufgetreten sein.

Der niedrigere Arbeitsspeicherverbrauch von Apache Flink resultiert aus dem aktiven Speichermanagement. Flink ist zwar wie Apache Spark eine Java-Anwendung, welche auf der *Java Virtual Machine (JVM)* ausgeführt wird, verlässt sich aber nicht ausschließlich auf den *JVM Garbage Collector*. Ein benutzerdefiniertier Speichermanager speichert Daten für die Verarbeitung in Byte-Arrays. Dies ermöglicht es, die Last auf einem Garbage Collector zu reduzieren.

Ein in absoluten Zahlen, großer Unterschied der Verarbeitungszeiten konnte nicht festgestellt werden. Dies liegt wahrscheinlich an der Kürze und dem ähnlichen Aufbau der Algorithmen selbst. Dennoch ist ein prozentual großer Vorsprung durch Apache Spark

zu erkennen, was wahrscheinlich an dem gut modellierten Pipeline Modell liegt.

Die Hypothese 1 konnte nicht aussagekräftig überprüft werden. Die Slaves waren in diesem Versuchsaufbau nicht komplett ausgelastet. Bei 3 involvierten Slaves hatte der 1. Slave bei Apache Flink einen Arbeitsspeicherverbrauch von durchschnittlich 4,9 Gigabyte. Die weiteren beiden Slaves benutzten nur noch zwischen 2,4 und 3,1 Gigabyte. Ähnlich sahen die Werte bei Apache Spark aus. Der erste Slave verbrauchte durchschnittlich 6,6 Gigabyte. Die Werte von Slave 2 und 3 lagen zwischen 4,6 und 4,8 Gigabyte. Das lässt darauf schließen, dass die Slaves mit dem vorliegenden Aufbau unterfordert waren. Dies ist ebenso an den Messwerten der Maxima der verarbeiteten Tweets pro Sekunde zu erkennen. Diese lagen bei beiden Anwendungen knapp unter dem Maximum, welches von der Kafka TwitterSource generiert werden konnte.

8 Ergebnisse der Arbeit und Ausblick

In diesem Kapitel werden die Ergebnisse dieser Thesis zusammengefasst. Abschnitt 8.1 enthält dafür eine Gegenüberstellung von Apache Flink und Apache Spark mit Aspekten, welche für die Erstellung einer Realtime-Erkennung von Hate-Speech auf Twitter relevant sind. Zum Schluss wird in Kapitel 8.2.1 ein Fazit gezogen und ein Ausblick gegeben, wie diese Arbeit noch fortgesetzt werden kann.

8.1 Flink vs Spark - Gegenüberstellung

Apache Flink und Apache Spark sind beides Projekte der Apache Software Foundation. Beide Frameworks sind dafür geeignet, komplexe Big-Data-Applikationen mit verschiedenen Arten der Verarbeitung umzusetzen. Dennoch unterscheiden sich die Frameworks im Detail betrachtet grundlegend. Für die Gegenüberstellung der beiden Frameworks, werden in den folgenden Abschnitten die wichtigsten Aspekte im Bezug auf einen Realtime Hate-Speech Filter hervorgehoben.

8.1.1 Vorverarbeitung

Apache Spark punktet durch die einfache Erstellung des Pipeline Modells. Dieses kann abgespeichert und jederzeit geladen werden. Dementgegen muss dies bei Apache Flink selbst implementiert und vor jedem Start erneut angelegt werden. Es ist bei Apache Flink zu empfehlen, Algorithmen für die Vorverarbeitung wie den TF-IDF Algorithmus, von anderen Bibliotheken zu importieren. Die einfache Textvorverarbeitung ist jedoch bei beiden Frameworks identisch.
Die Einfachheit der Spark Vorverarbeitung spiegelt sich auch in der Anzahl an Programmzeilen wieder. Apache Spark benötigt lediglich 125 Zeilen für die Vorverarbeitung. Apache Flink hingegen benötigt 317 Zeilen. Allgemein betrachtet sind beide Zahlenwerte

nicht besonders hoch. Dennoch liefern diese Werte einen ersten Eindruck von der Komplexität der schon in Spark enthaltenen Vorverarbeitungsalgorithmen, welche in Flink selbst implementiert werden müssten. Denn die Vorverarbeitung lässt sich noch durch viele Schritte erweitern.

Wie zu erkennen ist, liefert Apache Spark für den Schritt der Vorverarbeitung deutlich mehr Struktur und Hilfe als Apache Flink.

8.1.2 Stream-Verarbeitung

Apache Spark wurde von Anfang an für die Batch-Verarbeitung konzipiert, wohingegen Apache Flinks Grundkonzept schon immer die Datenstromverarbeitung war.
Beide Frameworks bieten dennoch durch verschiedene Mechanismen die Möglichkeit, Stream-Verarbeitungen umzusetzen. Es ist jedoch erneut hervorzuheben, dass Apache Spark technisch keine Stream-Verarbeitung, sondern eine Mini-Batch-Verarbeitung anbietet. Damit wird der Stream in kleinste Batches aufteilt, um ihn fortführend mit Apache Sparks großen Bibliotheken, welche auf der Batch-Verarbeitung basieren, weiterzuverwerten. Auch die Experimente spiegeln dies wieder. Apache Flink hat im Gegensatz zu Apache Spark eine deutlich niedrigere Latenz, da versucht wird, eintreffende Tweets direkt zu verarbeiten. Dies hebt die geringen Unterschiede in der Verarbeitungszeit eines Tweets im Vergleich zu Apache Spark auf. Will man also tatsächliche Stream-Verarbeitung mit niederiger Latenz und keine Mini-Batches, fällt die Wahl auf Apache Flink.

8.1.3 Klassifizierung

Wie auch die Vorverarbeitung, lässt sich die SVM von Apache Spark direkt in das Pipeline Modell integrieren und abspeichern. Somit kann beinahe die gesamte Hate-Speech Erkennung durch ein einfaches Modell dargestellt werden.

Das SVM Modell von Apache Flink lässt sich zwar nicht direkt in ein Pipeline Modell integrieren oder abspeichern, dennoch können die Gewichtsvektoren ausgegeben, hinterlegt und zugewiesen werden. Das größte Defizit bei Apache Flink ist jedoch, dass Flink ML rein auf die Batch-Verarbeitung ausgelegt ist. Somit gibt es vom Framework aus keine Möglichkeiten, Algorithmen des maschinellen Lernens in einem Streamingkontext zu verwenden. Dies ist extrem ausschlaggebend bei der Beurteilung des Frameworks an sich,

da die Arbeit mit einem vollwertigen Stream die Besonderheit und das Aushängeschild von Apache Flink selbst ist. Diese Punkte des fehlenden Pipeline Modells und der fehlenden Streamverarbeitung für das maschinelle Lernen lassen sich nur mit großem Aufwand kompensieren. Beinahe neunhundert Programmzeilen des StreamLine-Projektes [Ben] wurden für diesen Zweck integriert. Dementsprechend war die Umsetzung der Apache Flink Anwendung um einiges komplizierter, als die von Apache Spark. Bei Apache Spark ist nochmals positiv hervorzuheben, wie umfangreich die Bibliotheken (MLlib und Spark ML) des maschinellen Lernens sind. Anwendungen lassen sich Dank der gegebenen Implementationen wesentlich einfacher gestalten.

8.2 Fazit und Ausblick

Dieses Kapitel dient zum Zusammenfassen der Arbeit und ihrer Ergebnisse. Ebenso soll es einen Ausblick geben, wo man ansetzen könnte, um diese Arbeit fortzuführen.

8.2.1 Fazit

Die Experimente haben gezeigt, dass Apache Flink einen deutlichen Vorsprung im Bereich Streaming zu Apache Spark besitzt. Zum einen ermöglicht die wahre Stream-Verarbeitung von Apache Flink eine extrem niedrigere Latenz als Apache Sparks Structured Streaming. Zum anderen verbraucht Flink dabei weniger Arbeitsspeicher als sein Kontrahent. Dennoch konnte Apache Spark mit der einfachen Handhabung glänzen, denn wie sich aus dem Kapitel der Implementierungen (6) zeigt, bietet Apache Spark deutlich mehr Unterstützung bei der Erstellung der Hate-Speech Erkennung.
Dazu muss man sagen, dass Apache Flink seine Bibliothek des maschinellen Lernens seit Version 1.8.2 nicht weiter entwickelt hat. Wie bei Apache Spark soll in Zukunft die Table API von Flink, welche sich ähnlich wie Spark SQL verhält, Basis des maschinellen Lernens werden.
Die Vorteile von Apache Spark liegen in der Verarbeitung noch größerer Datenmengen, da sich durch die Anpassungen der Batchgröße ein wesentlich größerer Durchsatz erzeugen lässt.

Im Bezug auf eine Realtime Hate-Speech Erkennung auf Twitter empfielt es sich, Apache Flink zu verwenden. Die deutlich niedrigere Latenz ist dabei ausschlaggebend. Im Fall, dass die Anwendungen in Produktion auf Twitter verwendet werden, wäre es fatal, wenn Benutzer zu lange auf die Antwort warten müssten, ob der Tweet veröffentlicht werden darf oder nicht. Ebenso verbraucht die Anwendung von Apache Flink weniger Arbeitsspeicher, was wiederum weniger Kosten für das Unternehmen bedeutet. Das SVM und das Pipeline Modell sollten allerdings durch den Einsatz externer Bibliotheken abgebildet werden. Wie der Artikel *„A comparison on scalability for batch big data processing on Apache Spark and Apache Flink“* von Diego Garcia-Gil, aufgezeigt hat, ist die SVM von Apache Flink schlecht skalierbar. Für eine Hate-Speech Erkennung müssten riese Mengen an Datensätzen gelernt werden. Eine schlechte Skalierbarkeit der SVM wäre somit katastrophal und sie könnte so nicht in Produktion gehen.

8.2.2 Ausblick

Diese Arbeit kann als Grundlage für weitere Thesen dienen.
Die momentane Hate-Speech Erkennung ist auf die Erkennung von Hate-Speech im Text beschränkt. Hyperlinks werden ersetzt und nicht weiter betrachtet. Dies könnte jedoch ein wichtiger Punkt bei der Erkennung von Hate-Speech sein, da Texte mit Hyperlinks, welche zu hassverbeitenden Webseiten führen, ebenfalls als Hate-Speech klassifiziert werden sollten. Die Umsetzung davon könnte mit einer einfachen Liste klassifizierter Seiten oder dem Aufrufen und Klassifizieren der Seite in der Verarbeitungs-Pipeline durchgeführt werden. Ebenso werden Bilder von dem Algorithmus nicht verarbeitet. Für die Klassifizierung dieser könnte man neuronale Netze verwenden, um verbotene Symbole oder Zeichen zu erkennen.

Da die Table API von Apache Flink in Zukunft als Basis der maschinellen Lernalgorithmen dienen soll, wäre es nach ihrer Veröffentlichung interessant, wie sich Apache Flink speziell in den Punken Pipeline-Modell und Stream-Klassifizierung verbessert hat.

Da die Hypothese 1 (Kapitel 7.2.1) weder bestätigt noch wiederlegt werden konnte, empfielt es sich diese auf einem noch größeren Cluster mit einer viel größeren Menge an eintreffenden Daten zu überprüfen.

Literaturverzeichnis

[ABE+14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, December 2014. URL: https://doi.org/10.1007/s00778-014-0357-y, doi:10.1007/s00778-014-0357-y.

[And15] Jakob Smedegaard Andresen. Funktionale erweiterung des graphx frameworks, 2015.

[Bah19] Apache Bahir. Apache bahir. 2019. URL: https://bahir.apache.org/.

[Ben] Andras Benczur. D2. 1–real time stream mining library v1.

[Bot12] Léon Bottou. *Stochastic Gradient Descent Tricks*, pages 421–436. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. URL: https://doi.org/10.1007/978-3-642-35289-8_25, doi:10.1007/978-3-642-35289-8_25.

[CDE+16] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1789–1792, May 2016. doi:10.1109/IPDPSW.2016.138.

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing

in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[DRB10] Reinhold Decker Decker, Silvia Rašković, and Kathrin Brunsiek. *Diskriminanzanalyse*, pages 495–523. VS Verlag für Sozialwissenschaften, Wiesbaden, 2010.

[DWMW17] Thomas Davidson, Dana Warmsley, Michael Macy, and Ingmar Weber. Automated hate speech detection and the problem of offensive language. In *Proceedings of the 11th International AAAI Conference on Weblogs and Social Media*, ICWSM '17, 2017.

[Fac19] Facebook. Gemeinschaftsstandards. 2019. URL: https://www.facebook.com/communitystandards/objectionable_content.

[Fli19] Apache Flink. Apache flink: Scalable stream and batch data processing. 2019. URL: https://flink.apache.org.

[FN18] Paula Fortuna and Sérgio Nunes. A survey on automatic detection of hate speech in text. *ACM Computing Surveys (CSUR)*, 51(4):85, 2018.

[GGRGGH17] Diego García-Gil, Sergio Ramírez-Gallego, Salvador García, and Francisco Herrera. A comparison on scalability for batch big data processing on apache spark and apache flink. *Big Data Analytics*, 2(1):1, 2017.

[GS04] Edel Greevy and Alan F Smeaton. Classifying racist texts using a support vector machine. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 468–469. ACM, 2004.

[JST+14] Martin Jaggi, Virginia Smith, Martin Takác, Jonathan Terhorst, Sanjay Krishnan, Thomas Hofmann, and Michael I. Jordan. Communication-efficient distributed dual coordinate ascent. *CoRR*, abs/1409.1458, 2014. URL: http://arxiv.org/abs/1409.1458, arXiv:1409.1458.

[Kaf19] Apache Kafka. Apache kafka: A distributed streaming platform. 2019. URL: https://kafka.apache.org/.

[Kah19] Brewster Kahle. Internet archive. 2019. URL: http://archive.org.

[KKWZ15] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning spark: lightning-fast big data analysis*. O'Reilly Media, Inc., 2015.

[KPSA18] D. Kaushik, B. R. Prasad, S. K. Sonbhadra, and S. Agarwal. Post-surgical survival forecasting of breast cancer patient: A novel approach. In *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 37–41, Sep. 2018. doi:10.1109/ICACCI.2018.8554745.

[KZ18] Marc Kaepke and Olaf Zukunft. A comparative evaluation of big data frameworks for graph processing. In *4th International Conference on Big Data Innovations and Applications, Innovate-Data 2018, Barcelona, Spain, August 6-8, 2018*, pages 30–37, 2018. URL: https://doi.org/10.1109/Innovate-Data.2018.00012, doi:10.1109/Innovate-Data.2018.00012.

[Li19] Haifeng Li. Statistical machine intelligence and learning engine. 2019. URL: http://haifengl.github.io/smile/.

[MBY+16] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[MW15] David Meyer and FH Technikum Wien. Support vector machines. *The Interface to libsvm in package e1071*, page 28, 2015.

[NTT+16] Chikashi Nobata, Joel Tetreault, Achint Thomas, Yashar Mehdad, and Yi Chang. Abusive language detection in online user content. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pages 145–153, Republic and Canton of Geneva, Switzerland, 2016. International World Wide Web Conferences Steering Committee. URL: https://doi.org/10.1145/2872427.2883062, doi:10.1145/2872427.2883062.

[Red19] Redis. Redis. 2019. URL: https://redis.io/.

[RRC+17] Björn Ross, Michael Rist, Guillermo Carbonell, Benjamin Cabrera, Nils Kurowsky, and Michael Wojatzki. Measuring the reliability of hate speech annotations: The case of the european refugee crisis. *CoRR*, abs/1701.08118, 2017. URL: http://arxiv.org/abs/1701.08118, arXiv:1701.08118.

[SDC+16] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. Big data analytics on apache spark. *International Journal of Data Science and Analytics*, 1(3-4):145–164, 2016.

[SE19] Xiaowei Jiang Stephan Ewen, Fabian Hueske. Batch as a special case of streaming and alibaba's contribution of blink, Februar 2019. URL: https://flink.apache.org/news/2019/02/13/unified-batch-streaming-blink.html.

[Spa19] Apache Spark. Apache spark: Lightning-fast cluster computing. 2019. URL: https://spark.apache.org.

[SSZ13] Shai Shalev-Shwartz and Tong Zhang. Stochastic dual coordinate ascent methods for regularized loss minimization. *Journal of Machine Learning Research*, 14(Feb):567–599, 2013.

[TRH+15] Jonas Traub, Tilmann Rabl, Fabian Hueske, Till Rohrmann, and Volker Markl. Die apache flink plattform zur parallelen analyse von datenströmen und stapeldaten. In *LWA*, pages 403–408, 2015.

[TSSN15] M Trivedi, S Sharma, N Soni, and S Nair. Comparison of text classification algorithms. *International Journal of Engineering Research & Technology (IJERT)*, 4(02), 2015.

[Twi19a] Inc. Twitter. 2019. URL: https://developer.twitter.com/.

[Twi19b] Inc. Twitter. Die twitter regeln - sicherheit. 2019. URL: https://help.twitter.com/de/rules-and-policies/twitter-rules.

[Was16] Zeerak Waseem. Are you a racist or am I seeing things? annotator influence on hate speech detection on twitter. In *Proceedings of the First Workshop on NLP and Computational Social Science*, pages 138–142, Austin, Texas, November 2016. Association for Computational Linguistics. doi:10.18653/v1/W16-5618.

[XGFS13] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2484425.2484427, doi:10.1145/2484425.2484427.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[ZDL+12] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10–10. USENIX Association, 2012.

[ZXW+16] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016. URL: http://doi.acm.org/10.1145/2934664, doi:10.1145/2934664.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „- bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(ğ 18 Abs. 1 APSO-TI-BM bzw. ğ 21 Abs. 1 APSO-INGI)] - ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: ğ 16 Abs. 5 APSO-TI-BM bzw. ğ 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Realtime-Erkennung von Hate-Speech auf Twitter: Eine Evaluation von Apache Flink und Spark

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

_____	_____	_____
Ort	Datum	Unterschrift im Original