Prof. Dr. Markus Breunig & Stephan Schiffner *27. Februar 2018*

Einführung in Spark – ein Text Mining-Projekt

© Sergey Nivens / Fotolia.com

Apache Spark ist neben Apache Hadoop und Flink eine der bekanntesten – und momentan wahrscheinlich die verbreitetste – Anwendung für Clustercomputing, d. h. zur Verarbeitung großer Datenmengen. Dank umfangreicher Bibliotheken und APIs für Java, Scala und Python steht eine breite Palette an Funktionalität bereit, Module wie Spark Streaming und Spark SQL öffnen verschiedenste Anwendungsgebiete von real time-Monitoring zu tiefgehenden Datenanalysen. Eines der stärksten Features ist die In-Memory-Verarbeitung, die Anwendungen spürbar beschleunigt.

Spark löst damit viele Probleme, die Entwickler umtreiben: Effizienz, einfache Parallelisierung und Skalierbarkeit, Ausfallsicherheit und umfangreiche Module, und dazu noch eine API in der bevorzugten Programmiersprache. Im Folgenden stellen wir ein kleines Projekt vor, das Spark und einige der Werkzeuge aus der Spark-eigenen Machine Learning-Bibliothek MLlib verwendet, um eine intelligente Patentsuche zu realisieren.

Die Problemstellung – Gibt's da noch mehr zu?

Von Datenanalysten gerne eingesetzt wird Spark für ETL-Prozesse, interaktive Queries und Aufgaben aus dem Bereich maschinelles Lernen auf großen Datenmengen. Besonders relevant und interessant sind ETL-Aufgaben natürlich für nicht-strukturierte Daten, also etwa große Mengen Text, die in verschiedensten Dokumentenformen vorliegen können. Als anschauliches Beispiel stellen wir hier vor, wie man mit Hilfe von Spark die riesige Datenbank von US-Patenten durchsuchen kann. Und zwar wollen wir nicht einfach nach Stichworten suchen und filtern, sondern analytisch ein Stück weitergehen und solche Patente finden, die die größte Gemeinsamkeit zu, sagen wir, einer Beschreibung unseres eigenen geplanten Patents haben.

Dieses Vorgehen lässt sich natürlich auf ähnliche Problemstellungen übertragen, z. B. wenn wir gerade einen interessanten Artikel in einem Fachmagazin gelesen haben und nun ähnliche Artikel empfohlen haben möchten. Die Patentdaten werden aber von der US-Patentbehörde kostenfrei zur Verfügung gestellt und liegen als XML-Files zum Download bereit, was das Datensammeln etwas erleichtert. Pro Woche werden ca. 4.000 neue Patente in den USA zugelassen, in einem Jahr haben wir also etwa 200.000 Patenttexte, die wir durchsuchen wollen.

Beispiel für einen Patenttext: "*A docking connector to physically secure the portable media player; a communication interface to communicatively couple the portable media player to the docking station when the docking connector is physically securing the portable media player; and a projection module operably linked to the communication interface and configured to receive video information from the portable media player via the communication interface, and to project a video image derived from…*."

Im Folgenden werden wir zunächst kurz die Spark-Systemarchitektur und die grundlegenden Datenstrukturen vorstellen und dann direkt unsere intelligente Patentsuche entwickeln. Dabei verwenden wir einige nützliche Spark-Bibliotheken und zeigen nebenbei ein paar der

Autoren

Prof. Dr. Markus Breunig

Markus Breunig vermittelt businessrelevante Aspekte der Informatik an der Hochschule Rosenheim und ist bei Steadfor Senior Consultant… >> Weiterle

Stephan Schiffner

Stephan Schiffner ist Director Advanced Analytics bei Steadfor gesamtverantwortlich für Kunde Projekte und die strategische… >> Weiterlesen

Publikationen

Pragmatische Innovation

Techniken des Text Minings. Das sind Verfahren, um unstrukturiertem Text nützliche Informationen abzugewinnen.

Spark-Architektur und Datenstrukturen

Grundsätzlich baut Spark auf einer Master/Slave-Architektur auf, in der ein Master-Knoten (genannt Clustermanager) wirkt und Aufgaben auf die Slaves (genannt Worker-Knoten) verteilt. Der Spark-Driver ist ein Prozess, der den Spark Context bereitstellt und unser Zugang zur gesamten Spark-Engine. In unserem Beispiel läuft der Spark-Driver auf dem Master-Knoten (dies muss er nicht zwingend). Dazu benötigt Spark noch einen Clustermanager und ein verteiltes Speichersystem. Als Clustermanager stehen Hadoop YARN oder Mesos zur Auswahl, als Speichersystem HDFS, MapR-FS oder viele weitere. Zum Auspropieren und für Entwicklungszwecke gibt es Spark aber auch in einem lokalen Stand Alone-Modus, in dem Clustermanager und spezielles Dateisystem wegfallen können.

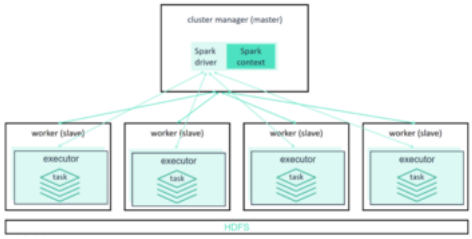

Abb. 1: Spark-Architektur. © Stephan Schiffner & Prof. Dr. Markus Breunig

Spark stellt drei grundlegende Datenstrukturen zur Verfügung: den resilient distributed dataset (RDD), den DataFrame (DF), und den DataSet (DS).

RDDs sind die älteste der Datenstrukturen. Wie der Name schon sagt, ist ein RDD eine über viele Knoten verteilte Datensammlung, man kann sie sich wie eine Collection von Records (streng typisiert) vorstellen, die verteilt über den Cluster gespeichert wird. Dadurch können Operationen parallelisiert und fehlertolerant (resilient) ausgeführt werden.

In neueren Spark-Versionen stehen zusätzlich DataFrames und DataSets zur Verfügung – beide kann man sich wie eine SQL-Tabelle vorstellen, bestehend aus Spalten, die Namen für den einfachen Zugriff darauf haben. Der Vorteil von DataSets gegenüber DataFrames ist, dass diese über Compilezeit-Typsicherheit verfügen. Diesen Vorteil können sie natürlich nur bei statisch typisierten Sprachen wie Java oder Scala ausspielen, für Python (das wir in diesem Artikel verwenden wollen), sind DataFrames die passende Abstraktion.

Alle drei Datenstrukturen sind unveränderlich (immutable). Neue Objekte werden entweder direkt aus einer Datenquelle wie einem Textfile eingelesen oder aus einer bestehenden Nicht-Spark-Datenstruktur (wie einer Liste oder einem Set) erzeugt.

Spark stellt vielfältige Operationen auf den RDDs/DFs/DSs zur Verfügung. Dabei ist zwischen Transformationen und Aktionen zu unterscheiden:

- Transformationen erzeugen aus einem RDD/DF/DS einen neuen RDD/DF/DS. Die Transformation *filter()* zum Beispiel erzeugt einen neuen RDD/DF/DS, der nur die Zeilen enthält, die eine angegebene Bedingung erfüllen; *withColumn()* wird verwendet, um eine neue Spalte zu einem DF/DS anzufügen, typischerweise abgeleitet aus den Werten einer oder mehrerer bestehender Spalten (d. h. es wird ein neuer DF/DS mit einer zusätzlichen Spalte zurückgegeben). Auch Joins/Unions/Projektionen und viele weitere Operationen sind möglich.
- Aktionen auf der anderen Seite berechnen einen "Wert", d. h. sie geben keinen RDD/DF/DS zurück. Beispiele dafür sind *first()* und *head()*, um ein oder einige wenige Elemente auszulesen, *show()* für einen tabellenformatierten Überblick eines DF oder *count(),* um die Anzahl Zeilen des RDD/DF/DS zu berechnen.

Der entscheidende Unterschied zwischen Transformationen und Aktionen ist, dass bei Transformationen die "lazy evaluation" angewendet werden, während Aktionen direkt berechnet werden: das Anwenden einer Transformation benötigt so gut wie keine Zeit, Spark merkt sich lediglich, wie der neue RDD/DF/DS aus dem Bestehenden berechnet werden soll – und das über beliebig viele Transformationen hinweg. Dadurch entsteht die sogenannte "lineage", also die Entstehungsgeschichte eines RDD/DF/DS. Erst das Ausführen einer Aktion bewirkt, dass diese Entstehungsgeschichte nun wirklich (verteilt über den Spark-Cluster) abgearbeitet wird.

Beim Abarbeiten der Entstehungsgeschichte/Lineage zeigt sich dann ein weiterer Unterschied zwischen RDD und DF/DS: bei DF/DS kommt ein Query-Optimizer (ähnlich dem einer relationalen Datenbank) zum Einsatz, bei RDDs nicht. Dadurch sind DF/DS den RDD häufig aus Performancegesichtspunkten deutlich überlegen.

Wir werden im Folgenden hauptsächlich mit DataFrames arbeiten (ein Umwandeln zwischen den Datenstrukturen ist jedoch jederzeit einfach möglich). Einige der Eigenschaften und

Unterschiede der drei Datenstrukturen sind in den folgenden Tabellen zusammengefasst.

Tabelle 1: Verfügbarkeit der verteilten Datenstrukturen in Spark

Datenstruktur	RDD	DF	DS
In einem Satz	Verteilte Sammlung von Datenelementen	Verteilte Tabelle mit benannten Spalten ("SQL-Tabelle" – Spark SQL)	Typsichere Erweiterung des DF
Seit Spark Version	1.0	1.3	1.6
Sprachen	Scala, Java, Python, R	Scala, Java, Python, R	Scala, Java

Tabelle 2: Eigenschaften der verteilten Datenstrukturen in Spark

Datenstruktur	RDD	DF	DS
Schema-Support	Nein	Ja	Ja
Typ-Sicherheit	Ja	Nein	Ja
Syntax-Fehler	Compile-Zeit	Compile-Zeit	Compile-Zeit
Analyse-Fehler	Compile-Zeit	Laufzeit	Compile-Zeit
Query-Optimization	Nein	Ja	Ja
Immutable	Ja	Ja	Ja
Lazy Evaluation	Ja	Ja	Ja

Sparkkontext und Datenlesen

Als Entwicklungsumgebung haben wir uns eine Stand Alone-Sparkumgebung eingerichtet, auf die wir mit einem Zeppelin-Notebook [2] zugreifen können. Als API verwenden wir PySpark, also die Python-Schnittstelle von Spark. Der Code liest sich hier besonders einfach, auch von Menschen, die wenig Python-Erfahrung haben. Die Daten für unser Projekt besorgen wir uns von den Servern des USPTO [3] und entpacken sie nach *data/patents/claims_2015.xml*. Damit sind unsere Vorbereitungen auch schon abgeschlossen.

Den Sparkkontext stellt uns Zeppelin standardmäßig schon in der Variablen *sc* bereit, genau wie übrigens die Spark-Shell, die interaktive Spark-Konsole. Andernfalls können wir den Sparkkontext auch per Hand konfigurieren und initialisieren.

```
from pyspark import SparkContext, SparkConf

conf = SparkConf().setAppName("Patentsuche").setMaster("local")

sc = SparkContext(conf=conf)
```

Um das SQL-Modul nutzen zu können, erstellen wir zunächst den erweiterten SQL-Kontext. Im Speziellen können wir jetzt unsere Daten in die zuvor beschriebenen DataFrames einlesen.

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
```

Der XML-Reader, den wir verwenden, erlaubt es uns, ein Root-Tag zu definieren, das als Wurzel des XML-Files betrachtet wird. Mit der Angabe des Row-Tags wird jedes *<patent>*-Element als eine Zeile des neuen DateFrames eingelesen. Ein Patent kann mehrere Kind-Elemente vom Typ *<claim>* haben, die wir in einer Gesamtbeschreibung in der Spalte "allClaims" konkatenieren. Um dann noch alle Patente eindeutig zu identifizieren, erhält jede Zeile eine ID. *withColumn()* erzeugt dabei jeweils eine neue Spalte im DataFrame. Einen Überblick über den DataFrame bekommen wir mit *show()* oder *head()*. *count()* dagegen zeigt erwartungsgemäß, wie viele Patente wir eingelesen haben. Die Lazy Evaluation von Spark sorgt dafür, dass in dem unten stehenden Code erst dann tatsächliche Berechnungen getriggert werden, wenn wir *count()* aufrufen. Die Transformationen davor definieren lediglich, wie der DataFrame berechnet werden soll. *count()* dagegen zählt zu den Aktionen, gibt also keinen neuen DataFrame, sondern einen Wert, in diesem Fall eben die Anzahl Zeilen, zurück.

```
from pyspark.sql.functions import monotonically_increasing_id, concat_ws
# Einlesen aller Patente von 2015
file_path = "data/patents/claims_2015.xml"
raw_df = sqlContext.read.format("com.databricks.spark.xml")
    .options(rootTag="patents", rowTag="patent")
    .load(file_path)
    .withColumn("allClaims", concat_ws(' ', 'claim'))
    .withColumn("id", monotonically_increasing_id())
    .drop("claim")

raw_df.count()
raw_df.head(2)
```

Datenaufbereitung und Speichern

Im nächsten Schritt wollen wir den Text so aufbereiten, dass der Fließtext auf einzelne, normalisierte Worte reduziert wird. Dafür verwenden wir zwei vorhandene Funktionen aus der MLlib-Library und zwei user-defined functions. Ein Tokenizer trennt den gesamten Claims-Text zunächst in einzelne Worte, getrennt an allen nicht-alphanumerischen Zeichen ("\W" in python regex-Syntax). Viele davon sind aber zu häufig und unspezifisch (*and*, *but*, *is*, *the*, etc.) und daher für unsere geplante Ähnlichkeitssuche nutzlos. Wir entfernen diese sogenannten Stoppworte mit dem StopWordsRemover in MLlib.

```
from pyspark.ml.feature import RegexTokenizer

tokenizer = RegexTokenizer(inputCol="allClaims", outputCol="claimToken", pattern="\\W")
tok_df = tokenizer.transform(raw_df)
```

```
from pyspark.ml.feature import StopWordsRemover

sw_remover = StopWordsRemover(inputCol="claimToken", outputCol="claimNoStopWords")
swr_df = sw_remover.transform(tok_df)
```

Zum Entfernen von Ziffern und Zahlen gibt es leider keine fertige Funktion, also definieren wir unsere eigene. Um eine Funktion auf einer Spalte des DataFrames anzuwenden, können wir user-defined functions (UDF) von SparkSQL nutzen. Wie in vielen SQL-Umgebungen erweitern sie auch in SparkSQL den Funktionsumfang der angebotenen SQL-Funktionalität. Wir wollen sie verwenden, um eine neue Spalte aus der Token-Spalte abzuleiten. Dazu entwerfen wir erst eine Funktion *remove_numbers()* in normaler Pythonsyntax, die wir in eine UDF umwandeln und auf die Token-Spalte anwenden. Der Code dazu sieht folgendermaßen aus:

```
import re
from pyspark.sql.functions import udf
from pyspark.sql.types import ArrayType, StringType

def remove_numbers(s):
            return list(word for word in s if not re.compile(r'^\d+$').match(word))
remove_numbers_udf = udf(remove_numbers, ArrayType(StringType()))
nn_df = swr_df.withColumn("claimNoNumbers", remove_numbers_udf("claimNoStopWords"))
```

Etwas komplizierter ist es, jedes Wort in eine Grundform zu überführen, so dass etwa electronical, electronics und electronic als der gleiche Wortstamm erkannt werden. Dafür verwenden wir einen Stemmer aus dem natural language toolkit nltk für Python. Wie oben müssen wir wieder eine Pythonfunktion für das eigentliche Stemmen definieren und in eine UDF umwandeln. Dann können wir unsere Funktion auf die inzwischen von Ziffern befreite Token-Spalte anwenden.

```
from nltk.stem.porter import *

stemmer = PorterStemmer()

def stem(in_vec):
    out_vec = []
    for t in in_vec:
        t_stem = stemmer.stem(t)
        if len(t_stem) > 2:
            out_vec.append(t_stem)
    return out_vec

stemmer_udf = udf(stem, ArrayType(StringType()))
stemmed_df = nn_df.withColumn("claimStemmed", stemmer_udf("claimNoNumbers"))
```

An dieser Stelle können wir den Zwischenstand sichern und die aufbereiteten Patenttexte auf die Festplatte schreiben. Das geht ganz einfach, zum Beispiel können wir den DataFrame mit allen relevanten Spalten im Parquet-Format speichern. Parquet ist dabei das Standardformat für das Schreiben von Dateien. Andere Optionen wie JSON, plain text oder auch JDBC stehen natürlich ebenso zur Verfügung. Das Speichern ist natürlich auch eine Aktion, d. h. hier wird die komplette Berechnung (Entstehungsgeschichte) durchgeführt und das Ergebnis "materialisiert".

```
tosave_df = stemmed_df.select(["id", "allClaims", "claimStemmed"])
tosave_df.write.mode("overwrite").save("data/stemmeddataframe.parquet")
```

Das Wiedereinlesen geht danach genauso einfach. Mit *df.show()* überprüfen wir noch einmal, dass auch das Schema, also die Spaltennamen der Spalten, wieder hergestellt wurden. Operationen auf diesem neu eingelesenen DataFrame müssen nicht mehr die ursprüngliche Lineage abarbeiten, sind also deutlich performanter.

```
df = spark.read.load("data/stemmeddataframe.parquet", format="parquet")
df.show()
```

Text Mining Basics – IDF und TF-IDF

Wir haben nun also jedem Patent eine Liste der darin vorkommenden gestemmten Worte (der "Terme") zugeordnet. Der noch verbleibende Schritt ist, ein Ähnlichkeitsmaß zwischen zwei Patenttexten zu definieren. Ein häufiges Vorgehen beim Definieren eines Ähnlichkeitsmaßes besteht darin, die Ausgangsobjekte in einen Euklidischen Vektorraum abzubilden, da es in einem solchen viele Möglichkeiten gibt, Distanzen bzw. Ähnlichkeiten zu berechnen (z. B. die Euklidische Distanz).

Abb. 2: Beispielpatent als Liste von Termen. © Stephan Schiffner & Prof. Dr. Markus Breunig

Eine einfache Option, aus Patenttexten Vektoren zu erzeugen, ist, die in allen Patenten zusammen vorkommenden Terme in beliebiger Reihenfolge anzuordnen, für jedes Patent zu zählen, wie oft jeder Term darin vorkommt, und diese Zahl als Wert der entsprechenden Dimension zu verwenden. Dies ist ein guter erster Schritt, da die Höhe des Wertes eines Terms (die term frequency, TF) darüber Aufschluss gibt, wie häufig der Term im Patent vorkommt, also vermutlich wie wichtig er für den Inhalt dieses Patent ist (vgl. Abb. 2: Beispielpatent als Liste von Termen und Abb. 3: TF (nicht normalisiert)). Allerdings sind wir damit noch nicht fertig.

Dokumentvektor

Wortvektor

	p1	p2	p3	p4	p5
electron	1	1	2	0	0
devic	1	1	1	1	1
compris	2	0	1	2	0
hous	1	1	1	0	1
front	1	1	0	0	0
portion	1	0	0	0	0
open	1	1	0	1	1
stem	0	0	0	3	4
cell	0	0	0	3	3
Summe	8	5	5	10	10

Wort-Dokument-Matrix
TF (nicht normalisiert)

Abb. 3: TF (nicht normalisiert). © Stephan Schiffner & Prof. Dr. Markus Breunig

Ein weiterer Aspekt, den wir berücksichtigen wollen, ist, dass Patente verschieden lang sind. Die TFs (Anzahl Vorkommen eines Terms) sind dadurch für ein langes Patent höher als für ein kurzes Patent, selbst wenn genau dieselben Terme in gleicher relativer Häufigkeit in den beiden Patenten vorkommen. Dies ist unschön, aber sehr leicht zu lösen: wir normieren den Vektor jedes Patents (z. B. so, dass die darin enthaltenen TFs in Summe 1 ergeben). Damit erhalten wir die normierte term frequency (vgl. Abb. 4: TF (normalisiert))

	p1	p2	p3	p4	p5
electron	0,125	0,2	0,4	0	0
devic	0,125	0,2	0,2	0,1	0,1
compris	0,25	0	0,2	0,2	0
hous	0,125	0,2	0,2	0	0,1
front	0,125	0,2	0	0	0
portion	0,125	0	0	0	0
open	0,125	0,2	0	0,1	0,1
stem	0	0	0	0,3	0,4
cell	0	0	0	0,3	0,3
Summe	1	1	1	1	1

TF (normalisiert)

Abb. 4: TF (normalisiert). © Stephan Schiffner & Prof. Dr. Markus Breunig

Weiterhin beobachten wir, dass manche Terme in sehr vielen Patenten vorkommen (z. B. wird der Term "claim" in fast jedem Patent vorkommen, da er nicht in unserer Stoppwortliste enthalten ist), und uns somit nur wenig helfen, die Patente zu

unterscheiden, während andere Terme in nur wenigen Patenten vorkommen, also vermutlich ein starkes Anzeichen für die Ähnlichkeit dieser Patente und den Unterschied zu allen anderen Patenten sind. Wir definieren die IDF oder *inverse document frequency* eines Terms als *log(N/n)*, wobei *N* die Anzahl aller Patente ist, *n* die Anzahl von Patenten in denen der Term vorkommt. Terme die in fast allen Patenten vorkommen, haben eine IDF von nahezu 0. In umso weniger Patenten ein Term vorkommt, desdo höher ist sein IDF-Wert (vgl. Abb. 5: IDF-Werte).

	p1	p2	p3	p4	p5	# Patente	IDF
electron	0,125	0,2	0,4	0	0	3	0,51
devic	0,125	0,2	0,2	0,1	0,1	5	0,00
compris	0,25	0	0,2	0,2	0	3	0,51
hous	0,125	0,2	0,2	0	0,1	4	0,22
front	0,125	0,2	0	0	0	2	0,92
portion	0,125	0	0	0	0	1	1,61
open	0,125	0,2	0	0,1	0,1	4	0,22
stem	0	0	0	0,3	0,4	2	0,92
cell	0	0	0	0,3	0,3	2	0,92
Summe	1	1	1	1	1		

TF (normalisiert)

Abb. 5: IDF-Werte. © Stephan Schiffner & Prof. Dr. Markus Breunig

Den finalen TF-IDF-Vektor für jedes Patent erzeugen wir, indem wir jeden (normierten) TF-Wert des Patentes mit dem IDF-Wert des Terms multiplizieren, d. h. den Term mit seiner "Unterscheidungskraft" bewerten (vgl. Abb. 6: TF-IDF-Matrix).

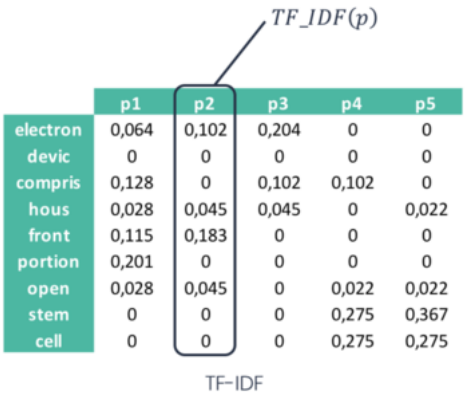

Abb. 6: TF-IDF-Matrix. © Stephan Schiffner & Prof. Dr. Markus Breunig

Alle Berechnungen können wir leicht mit den Funktionen durchführen, die uns MLlib bereitstellt. Zunächst zählen wir, welche Terme wie oft in jedem Patent vorkommen und speichern diese Information als Vektor für das Patent. Diese counts teilen wir durch die Anzahl der Terme des Patents, normalisieren also diesen Vektor. Die IDF berechnen wir aus den term frequencies. Dazu wird gezählt, in wie vielen der Dokumentvektoren der count eines Terms ungleich 0 ist und dieser Term entsprechend gewichtet.

CountVectorizer() muss hierbei zweimal über den DataFrame iterieren. Einmal, um herauszufinden, welche Worte insgesamt vorkommen und einen entsprechend dimensionierten Vektor bereitzustellen, was mit der Funktion *fit()* geschieht, und ein zweites Mal, um per *transform()* die Gewichte zu setzen. Genauso muss *IDF()* zuerst die IDF-Gewichte jedes Wortes per *fit()* finden, um sie dann per *transform()* auf die Dokumentvektoren anzuwenden, also mit der TF zu multiplizieren.

```
from pyspark.ml.feature import CountVectorizer
from pyspark.ml.feature import Normalizer
from pyspark.ml.feature import IDF

cv = CountVectorizer(inputCol="claimStemmed", outputCol="claimTF", vocabSize=vSize)
cvModel = cv.fit(df)
tf_df = cvModel.transform(df)

normalizer = Normalizer(inputCol="claimTF", outputCol="claimTFN")
tfn_df = normalizer.transform(tf_df)

idf = IDF(inputCol="claimTFN", outputCol="IDF")

idfModel = idf.fit(tfn_df)
tfidf_df = idfModel.transform(tfn_df)
```

Die Tokens mit den höchsten TF-IDF-Werten identifizieren in der Regel aussagekräftige Schlagworte. Beispiele für einige zufällig gewählte Patente und die besten Tokens, die aus den Patentbeschreibungen extrahiert wurden, sehen wir hier:

Tabelle 3: Die besten Tokens aus den Patentbeschreibungen

Patent	Beste TF-IDF Token
Tragbarer Bewegungsmelder	position, portable, device, movement, indicating
Tragbarer Videoprojektor	portable, station, media, video, projection
Neue Pflanzenzüchtung	plant, unique, new, combination, substantially

Kosinus-Ähnlichkeit für Dokumente

Eigentlich sind wir nun fertig, und könnten einfach die Euklidische Distanz der TF-IDF-Vektoren als Maß für die Unterschiedlichkeit der Patente verwenden. Dies hat jedoch zwei

Nachteile: zum einen ist es eine Distanz und keine Ähnlichkeit – kleine Werte (nahe bei 0) bedeuten hohe Ähnlichkeit, große Werte geringe Ähnlichkeit. Zum anderen sind unsere TF-IDF-Vektoren sehr lang, d. h. der Vektorraum ist sehr hochdimensional: es gibt so viele Dimensionen, wie es verschiedene Terme in allen Patenten zusammen gibt. Dies sind knapp 100.000! Die Euklidische Distanz leidet unter dem "curse of dimensionality", d. h. je höherdimensional der Vektorraum wird, umso weniger aussagekäftiger ist sie (die Distanzen werden sich immer ähnlicher).

Beide Probleme können wir jedoch elegant lösen, indem wir an Stelle der Euklidischen Distanz die Kosinus-Ähnlichkeit verwenden. Wenn wir uns die TF-IDF-Vektoren als "Pfeile" vom Ursprung aus vorstellen, beschreibt die Richtung des Pfeils das inhaltliche Thema des Patents. Der Winkel zwischen zwei solchen "Patent-Pfeilen" beschreibt, wie ähnlich sich die beiden Patente sind – ist der Winkel 0 Grad oder nahe an 0, gehen die Patente in die gleiche Richtung, ist er 90 Grad oder nahe daran, sind die Patent so unterschiedlich wie möglich. Der Winkel liegt also zwischen 0 und (absolut) 90 Grad (vgl. Abb. 7: Patente und Winkel).

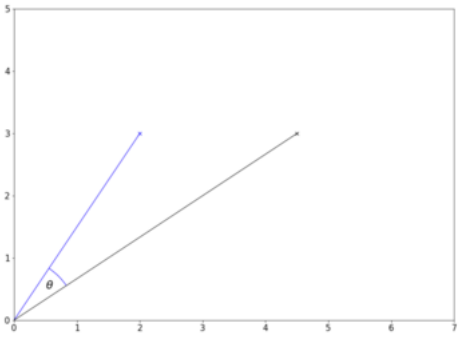

Abb. 7: Patente und Winkel. © Stephan Schiffner & Prof. Dr. Markus Breunig

Wenn wir auf diesen Winkel noch die Kosinusfunktion anwenden, erhalten wir eine Zahl zwischen 1 = cos(0 Grad) bei maximaler Ähnlichkeit und 0 = cos(90 Grad) bei maximaler Unähnlichkeit (vgl. Abb. 8: Kosinusfunktion). Dies ist ein sehr schönes Ähnlichkeitsmaß, und lässt sich noch dazu sehr einfach berechnen (zur Erinnerung aus der Mathematik: der Kosinus zweier Vektoren eines Euklidischen Vektorraums ist das Skalarprodukt der beiden geteilt durch das Produkt der Normen).

Abb. 8: Kosinusfunktion. © Stephan Schiffner & Prof. Dr. Markus Breunig

Wir benötigen zwei Funktionen: Eine user-defined function zur Berechnung der Kosinus-Ähnlichkeit zweier Vektoren und eine Funktion, welche die Ähnlichkeit zwischen einem Suchtext und allen existierenden Patenten zurückgibt. Als Suchtext wählen wir dabei eines der Patente in unserem Datensatz aus.

```
from pyspark.sql.types import FloatType

# Berechne die cosine-simliarity: dot product / product der L2-Norms
def csim(vec1, vec2):
    return vec1.dot(vec2)/vec1.norm(2)*vec2.norm(2))

# Berechne die Ähnlichkeiten zu einem gegebenen Patent
def getSimilarities(df, id):
    patent = df.where("id == "+id)
    my_tfidf = patent.select("IDF").first().IDF
    csim_udf = udf(lambda x: round(float(csim(my_tfidf, x)),2))
    sim_df = df.withColumn("similarity", csim_udf("IDF"))
    return sim_df
```

Suche nach ähnlichen Patenten

Der DataFrame *tfidf_df* enthält für jedes Patent einen TF-IDF-Vektor in der Spalte IDF. Unsere Funktion *getSimilarities()* berechnet die Ähnlichkeit zwischen einem ausgewählten Patenttext und allen weiteren Texten im DataFrame (in einer Spalte similarity). Um die relevanten, besonders ähnlichen Patente zu finden, müssen wir also nur noch das Ergebnis von *getSimilarities()* nach der errechneten Ähnlichkeit sortieren. Mit *show()* zeigen wir uns die besten 4 Treffer an.

```
from pyspark.sql.functions import desc

getSimilarities(tfidf_df, "10")
.select(["id", "similarity", "allClaims"])
    .orderBy(desc("similarity"))
    .show(4, truncate=100)
```

Der erste Treffer ist natürlich unser Suchtext selbst, mit einer Kosinus-Ähnlichkeit von 1.0. Er beschreibt einen tragbaren Videoplayer mit Dockingstation. Die nächsten beiden Ergebnisse, mit ähnlicher Bewertung knapp unter 0.6, beschreiben ebenfalls tragbare Elektronikgeräte mit Dockingstation. Schon das dritte Patent dagegen handelt nicht von elektronischen Geräten, sondern von einem parfümierten Luftbefeuchter. Die errechnete Ähnlichkeit ist auch deutlich geringer und ergibt sich daraus, dass auch der Luftbefeuchter modular zusammengesetzt ist und in der Beschreibung das Wort "docking station" verwendet wird.

Tabelle 4: Die besten 4 Treffer

ID	similarity	allClaims
10	1.00	a docking connector to physically secure the portable media player; a communication interface to communicatively couple the portable media player to the docking station...
3941	0.58	a base at least partially configured to support the electronic device; an electrical connector coupled to the base and configured to receive the connector port of the electronic device...
1243	0.56	a docking tray adapted for seating a portable electronic device therein with a docking port of the portable electronic device seated against an interface portion thereof, the interface...
802	0.36	a docking station; a substance storage medium having a substance to be dispensed; a receiver housing carried by said docking station for receiving said storage medium in a docked operating...

Wir können natürlich nicht per Hand alle Suchergebnisse überprüfen. Aber zumindest ein Überblick über die Verteilung aller Distanzen hilft uns, die Suchergebnisse zu validieren. Dazu berechnen wir die Ähnlichkeiten und transformieren den DataFrame in einen RDD, der nur noch die Zahlenwerte enthält. Dieses lassen wir uns mit *collect()* als Python-Liste zurückgeben. Zum Plotten eines Histogramms verwenden wir matplotlib. Wir erkennen einige wenige Ergebnisse mit hoher Ähnlichkeit, von denen wir schon gesehen haben, dass sie recht gut zu unserem Suchtext passen. Der weitaus größte Teil hat offensichtlich eine geringe Ähnlichkeit zu "unserem" Patent.

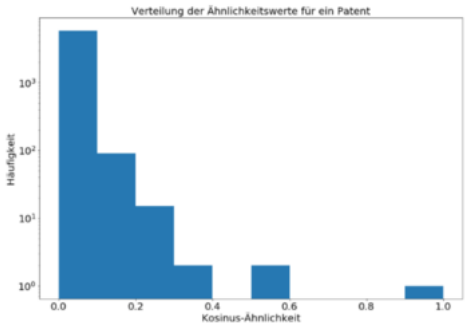

Abb. 9: Verteilung der Ähnlichkeitswerte für ein Patent. © Stephan Schiffner & Prof. Dr. Markus Breunig

```
import matplotlib.pyplot as plt

similarities = getMostSimilar(tfidf_df, "10")
.select(["similarity"])
.rdd.map(lambda row: float(row['similarity'])).collect()

plt.hist(similarities)
plt.yscale('log', nonposy='clip')
plt.title("Verteilung der Ähnlichkeitswerte für ein Patent"
plt.xlabel("Kosinus-Ähnlichkeit")
plt.ylabel("Häufigkeit")

plt.show()
```

Take away

Spark gehört inzwischen zum normalen Werkzeugkasten von Datenanalysten, Data Engineers und Data Scientists. Seine Vorteile liegen darin, auch auf großen Datenmengen schnell und einfach Analysen durchzuführen und durch interaktive Queries einen besseren Einblick in Daten zu bekommen. Nach einem kurzen Überblick über Architektur und Datenstrukturen haben wir an unserem Einführungs-Projekt gesehen, wie eine Mini-Applikation von ETL zu Analyse aufgebaut werden kann. Nebenbei haben wir noch einige grundlegende Techniken zur Textvorverarbeitung und Methoden des Text Minings kennengelernt. Für Spark gibt es aber für fast jeden erdenklichen Zweck eine (frei verfügbare!) Bibliothek oder Funktion, so dass der Kreativität des Analysten kaum Grenzen gesetzt sind.

Quellen

[1] US-Patentbehörde

[2] Zeppelin-Notebook
[3] USPTO

Autoren

Prof. Dr. Markus Breunig

Markus Breunig vermittelt businessrelevante Aspekte der Informatik an der Hochschule Rosenheim und ist bei Steadforce als Senior Consultant im Bereich Data Analytics tätigt und Managing Partner der go3consulting PartG.
>> Weiterlesen

Stephan Schiffner

Stephan Schiffner ist Director Advanced Analytics bei Steadforce und gesamtverantwortlich für Kunden-Projekte und die strategische Weiterentwicklung des Bereichs.
>> Weiterlesen

Publikationen

- Pragmatische Innovation

Das könnte Sie auch interessieren

Was macht eigentlich ein Bundler?

IDs – wie Äpfel und Birnen?

Java 17 – Lieblingsfeatures

Artefakte auf Maven Central veröffentlichen

Was ist funktionale Programmierung?

Einstieg in Python

Kommentare (0)

Neuen Kommentar schreiben

Name:

E-Mail-Adresse:

Kommentar:

Ich verstehe und akzeptiere die Datenschutzbestimmungen.

Absenden