Beschreibung der Aufgabe

Um beispielsweise häufig genutzte Wörter in einem umfangreichen Textkorpus schnell zu identifizieren, werden oft verteilte, gut skalierbare Frameworks wie Apache Hadoop in Kombination mit Java 8 eingesetzt. Der vorliegende Bericht beschäftigt sich mit der Aufgabe, die Häufigkeit von Wörtern in vorgegebenen Texten zu berechnen und eine TOP-10-Liste dieser Wörter zu erstellen.

a.) Berechnen Sie die Vorkommenshäufigkeit für alle Wörter in gegebenen Texten

b.) Stellen Sie die Ergebnisse als TOP-10 Liste getrennt für jede Sprache vor, mit Ausnahme der Stoppwörter (z.B. “er”, “sie”, “es”, ..)

Der verwendete Datensatz umfasst Bücher, die in deutscher, englischer, französischer, italienischer, niederländischer, russischer, spanischer und ukrainischer Sprache verfasst wurden. Insgesamt stehen 372 Bücher mit einem Textvolumen von 134 MB zur Verfügung.

Um die Vorteile von Hadoop umfassend zu testen, wurden auch Tests mit größeren, aus den einzelnen Büchern und Sprachen zusammengestellten Textkorpussen durchgeführt. Hierbei wurden alle Wörter einer Sprache zu einem einheitlichen Textkorpus zusammengeführt und evaluiert (siehe obige Aufgaben).

Darüber hinaus sollten sogenannte Stoppwörter aus den Texten entfernt werden, um sie nicht in die TOP-10-Liste aufzunehmen. Zu diesem Zweck wurden öffentlich verfügbare Quellen einzelner Stoppwortlisten in eine einzige, nach Sprache durchsuchbare Liste überführt. Diese Liste umfasst 58 verschiedene Sprachen und 25.228 Stoppwörter.

Neben den oben genannten Aufgaben sollen ebenfalls Parameter zur Verbesserung der Laufzeit in Hadoop ermittelt werden. Dies könnte durch die Analyse von Faktoren wie die Anzahl der Mapper und Reducer, die Konfiguration der Cluster-Ressourcen, die Optimierung der Datenverteilung und die Minimierung von I/O-Operationen erfolgen. Die Identifizierung und Anpassung dieser Parameter kann signifikante Auswirkungen auf die Effizienz und Geschwindigkeit der Verarbeitung von MapReduce-Aufgaben haben.

Lösungsbeschreibung

Die vorliegende Aufgabe lässt sich in zwei Schritte unterteilen:

- Zählen der Wörter
- Sortieren der Wörter anhand der gezählten Anzahl

Zählen der Wörter

Um die einzelnen Wörter eines Textes zu zählen, wird das Map-Reduce-Prinzip des Hadoop-Frameworks eingesetzt. Zuerst werden die entsprechende Textdatei und die Sprache des Textes angegeben, gefolgt von der Hinzufügung der Stoppwortliste. Schließlich wird der Ausgabeort spezifiziert. Diese Aufgabe kann auf mehrere Map-Reducer-Prozesse gleichzeitig verteilt werden, um die Durchlaufzeiten zu reduzieren.

Sobald ein Mapper den Text erhält, werden zunächst die Wortgrenzen ermittelt und die Wörter in Kleinbuchstaben umgewandelt. Je nach Sprache wird jedes Wort in der Stoppwortliste gesucht. Sollte das Wort nicht in der Liste vorhanden sein, wird ein Tupel-Eintrag mit dem Wort und der Zahl 1, z. B. („wort“, 1), erstellt. Um die Stoppwortsuche möglichst effizient zu gestalten, wird ein HashSet mit O(1) Lookup verwendet.

Im Reduce-Prozess werden dann die erstellten Tupel aus Wort und Anzahl für jedes einzelne Wort aufsummiert und schließlich ausgegeben. Die Ausgabe sieht etwa so aus: [("wort", 2030), ("wortarm", 1), ...]. Somit erhalten wir eine Liste an Tupeln mit Wörtern und deren Häufigkeit im Text.

Sortieren der Wörter anhand der gezählten Anzahl

Diese Liste soll nun nach der Häufigkeit der Wörter sortiert werden. Zuerst werden wieder die Ausgangsdatei und der Speicherort angegeben. Im Mapping wird der eingelesene Text, bestehend aus Wort und Häufigkeit, genutzt, um den Schlüssel (Anzahl, Wort) zu erzeugen. Dieser Schlüssel kann zuerst nach der Anzahl und, falls die Anzahl gleich ist, danach nach dem Wort sortiert werden.

Die so sortierbare Menge an Schlüsseln wird an einen Reducer übergeben, der die korrekte Sortierung sicherstellt. Ausgegeben wird erneut eine Liste an Tupeln, diesmal nach Häufigkeit und dann Wort sortiert, z. B.: [(5683, rief), (5610, hand), ..., (1355, standen), (1355, tür), ...].

Quellcode Ausschnitte

Nachfolgend werden einige interessante Quellcode Ausschnitte gezeigt und erklärt.

Map() Funktion des WordCountMapper:

In dieser Funktion werden einzelne Wörter anhand des regulären Ausdrucks (RegEx) `[\p{L}\d]+` identifiziert. Hierbei steht `\p{L}` für einen Buchstaben einer beliebigen Sprache und `\d` beschreibt eine beliebige Zahl (0-9). Die eckigen Klammern `[...]` repräsentieren eine Auswahl aus diesen Ausdrücken, gefolgt von `+`, was bedeutet, dass ein oder mehrere dieser Ausdrücke hintereinander erwartet werden. Wenn ein anderes Zeichen, wie beispielsweise ein Leerzeichen, erkannt wird, gilt die vorhergehende Menge an Zeichen als abgeschlossen und wird als Wort gewertet.

```
protected void map(Object key, Text value, Mapper<Object, Text, Text,
IntWritable>.Context context) throws IOException, InterruptedException {
    Pattern pattern = Pattern.compile("[\\p{L}\\d]+");
    Matcher matcher = pattern.matcher(value.toString());
    while (matcher.find()) {
        String word = matcher.group().toLowerCase(Locale.ROOT);
        if (stopwords.contains(word))
            continue;
        context.write(new Text(word), new IntWritable(1));
        context.getCounter("WordCount", "TotalWords").increment(1);
    }
}
```

Der Vergleich, ob Wörter in der Stoppwortliste vorhanden sind, sowie das Zählen erfolgt in einer kleingeschriebenen Variante des Wortes. Dadurch werden beispielsweise Verben oder Adjektive am Satzanfang gleich gezählt wie solche, die mitten im Satz stehen. Durch das Zählen der Wörter und das Stoppen der Ausführungszeit können beispielsweise Wörter pro Minute berechnet werden.

Map() Funktion des SortByCountMapper:

In dieser Funktion wird aus dem eingelesenen Text das Wort und die Anzahl, getrennt durch einen Tabulator, extrahiert und in den `CompositeKey` überführt.

```
public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
    //Input format: word\tcount
    String[] parts = value.toString().split("\t");
    if (parts.length == 2) {
        String word = parts[0];
        int count = Integer.parseInt(parts[1]);
        compositeKey = new CompositeKey(count, word);
        context.write(compositeKey, NullWritable.get());
        context.getCounter("KeyCount", "TotalKeys").increment(1);
        return;
    }
        throw new IOException("Invalid input format: " +
value.toString());
}
```

Durch das Zählen der Anzahl der Schlüssel und das Stoppen der Ausführungszeit kann auch hier die Anzahl der Schlüssel pro Minute berechnet werden.

Ergebnisse/Zwischenergebnisse

Um die Wörter tatsächlich zu zählen und zu sortieren, können die folgenden Befehle verwendet werden. Durch das Anpassen der Parameter in den Befehlen können beliebige Textdateien in unterschiedlichen Sprachen eingelesen und verarbeitet werden.

Zählen der Wörter

```
bin/hadoop jar fs/hadoop_wordcount.jar de.floriansymmank.WordCountDriver
de /data/de/de_all.txt /output/de_all_wordcount /data/stopwords.json
/output/stats.txt
```

Dieser Befehl zählt die Wörter in der Datei `/data/de/de_all.txt`, indem der Job im `WordCountDriver` ausgeführt wird. Dieser speichert das Ergebnis in der Datei `/output/de_all_wordcount`. Der Parameter `de` definiert die Sprache des Texts, was bedeutet, dass in der Stoppwortliste nur nach deutschen Stoppwörtern gesucht wird. Sollte der Sprachparameter nicht mit einer der Sprachen in der Stoppwort-Datei übereinstimmen, werden die Stoppwörter nicht entfernt.

Daraus ergibt sich die Ausgabe:

```
Input File: de_all.txt
Input File Size (bytes): 34102705
Language: de
Total Words: 2164266
Elapsed Time (ms): s
Words per Minute: 7401309
```

Die Datei `part-r-00000` im Ordner `/output/de_all_wordcount` enthält dann diese Werte:

```
...
innenräume  1
innenschau  2
innenseite  5
innenstadt  1
...
```

Sortieren der Wörter anhand der gezählten Anzahl

```
bin/hadoop jar fs/hadoop_wordcount.jar
de.floriansymmank.SortByCountDriver
/output/de_all_wordcount/part-r-00000 /output/de_all_sorted
```

Dieser Befehl sortiert die Wörter in der Datei `/output/de_all_wordcount/part-r-00000` anhand ihrer Häufigkeit und speichert das Ergebnis in `/output/de_all_sorted`. Die Ausgabe liefert Folgendes::

```
Output File: de_all_sorted
Input File: part-r-00000
Input File Size (bytes): 2166304
Total Keys: 157655
Elapsed Time (ms): 14750
Keys per Minute: 641308
```

In der Datei `/output/de_all_sorted` steht dann Folgendes:

```
5683  rief
5610  hand
5172  augen
...
```

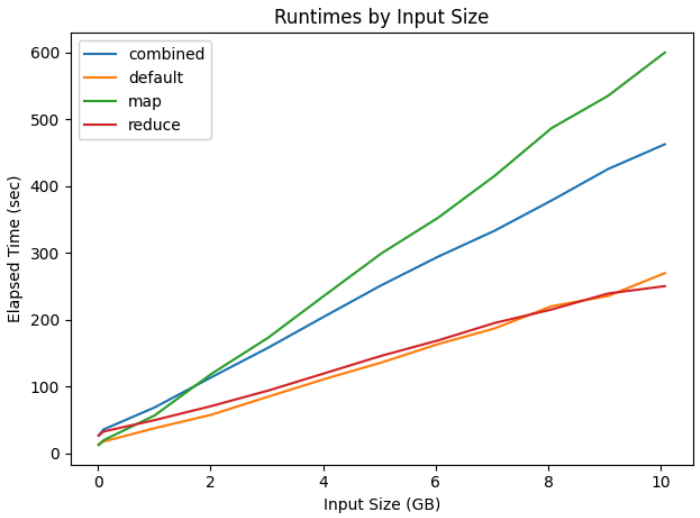
Parameter Hadoop

Im nachfolgenden Abschnitt wurden einige Parameter des Hadoop-Frameworks angepasst, um die Laufzeit zu reduzieren. Die Tabelle beschreibt die jeweiligen Einstellungen für die verschiedenen Serien. Es wurden spezifische Anpassungen für jede Phase vorgenommen.

Parameter Name/Series	Default	Map	Reduce	Combined
NumReduceTasks	1	1	8	8
mapreduce.reduce.memory.mb	1024	1024	4096	4096
mapreduce.reduce.java.opts	-Xmx768m	-Xmx768m	-Xmx3072m	-Xmx3072m
mapreduce.reduce.speculative	false	false	true	true
mapreduce.output.compress	false	false	true	true
mapreduce.output.compress.codec	Not set	Not set	org.apache.hadoop.io.compress.GzipCodec	org.apache.hadoop.io.compress.GzipCodec
mapreduce.map.memory.mb	1024	2048	1024	2048
mapreduce.map.java.opts	-Xmx768m	-Xmx1536m	-Xmx768m	-Xmx1536m
mapreduce.map.speculative	false	true	false	true
mapreduce.task.io.sort.mb	100	512	100	512
mapreduce.task.io.sort.factor	10	100	10	100

Die untersuchten Dateigrößen der Inputdatei sind: 10MB, 100MB, 1GB, 2GB, 3GB, 4GB, 5GB, 6GB, 7GB, 8GB, 9GB und 10GB.

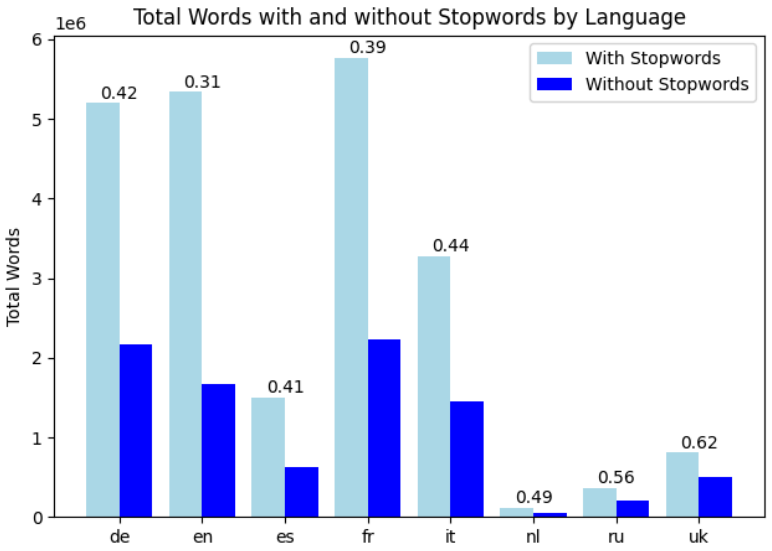
Das Diagramm stellt die vier Serien: Default, Map, Reduce und Combined dar.



Ein flacher Kurvenverlauf deutet eine schnellere Laufzeit an. Die Konfiguration in der Map-Serie verlangsamt die Laufzeit sichtlich, während die Reduce-Serie mit zunehmender Dateigröße näher an die Laufzeit der Default-Serie heranreicht und bei größeren Dateien gleich oder etwas schneller ist. Die kombinierte Serie ist zwar schneller als die Map-Serie, erreicht jedoch deutlich nicht die Werte der Reduce- oder Default-Serie.

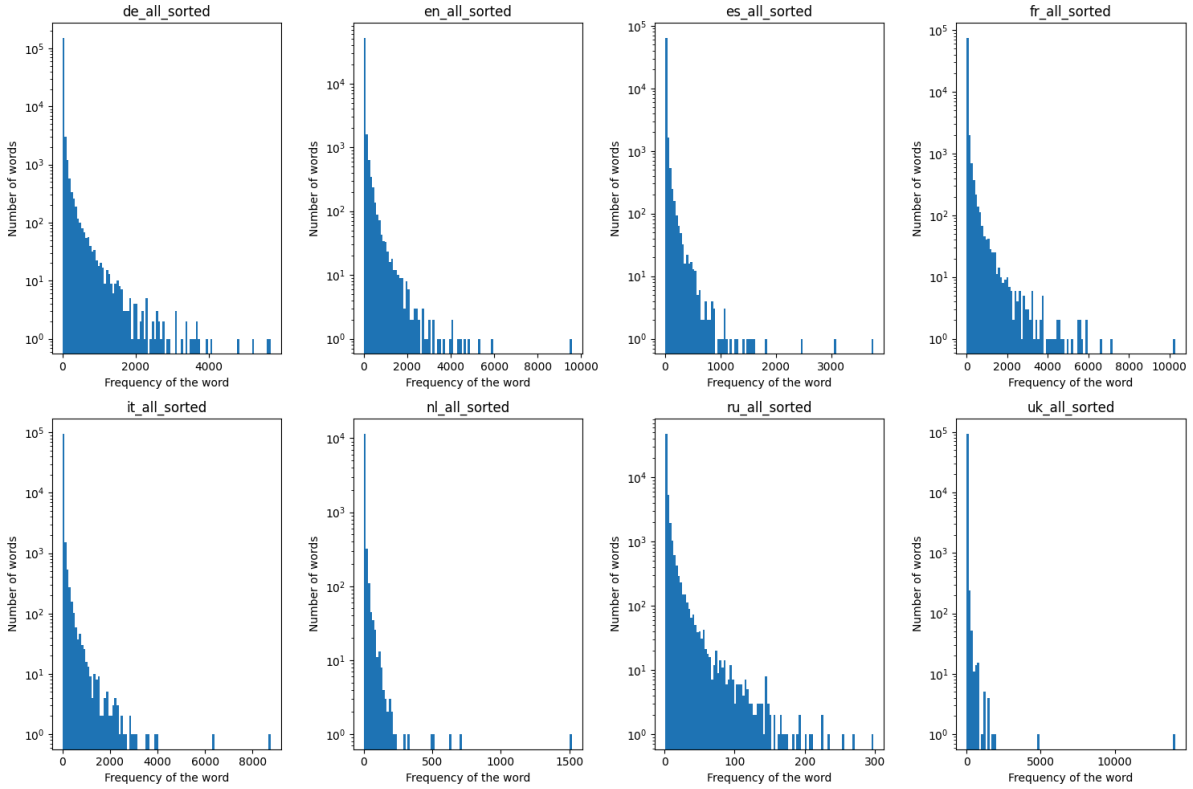
Diagramme

Das folgende Diagramm zeigt das Verhältnis der Wortanzahl in verschiedenen Sprachen, sowohl vor als auch nach der Entfernung von Stoppwörtern. Darüber hinaus wird das Verhältnis von $\frac{Text\ ohne\ Stoppwörter}{Text\ mit\ Stoppwörter}$ dargestellt.



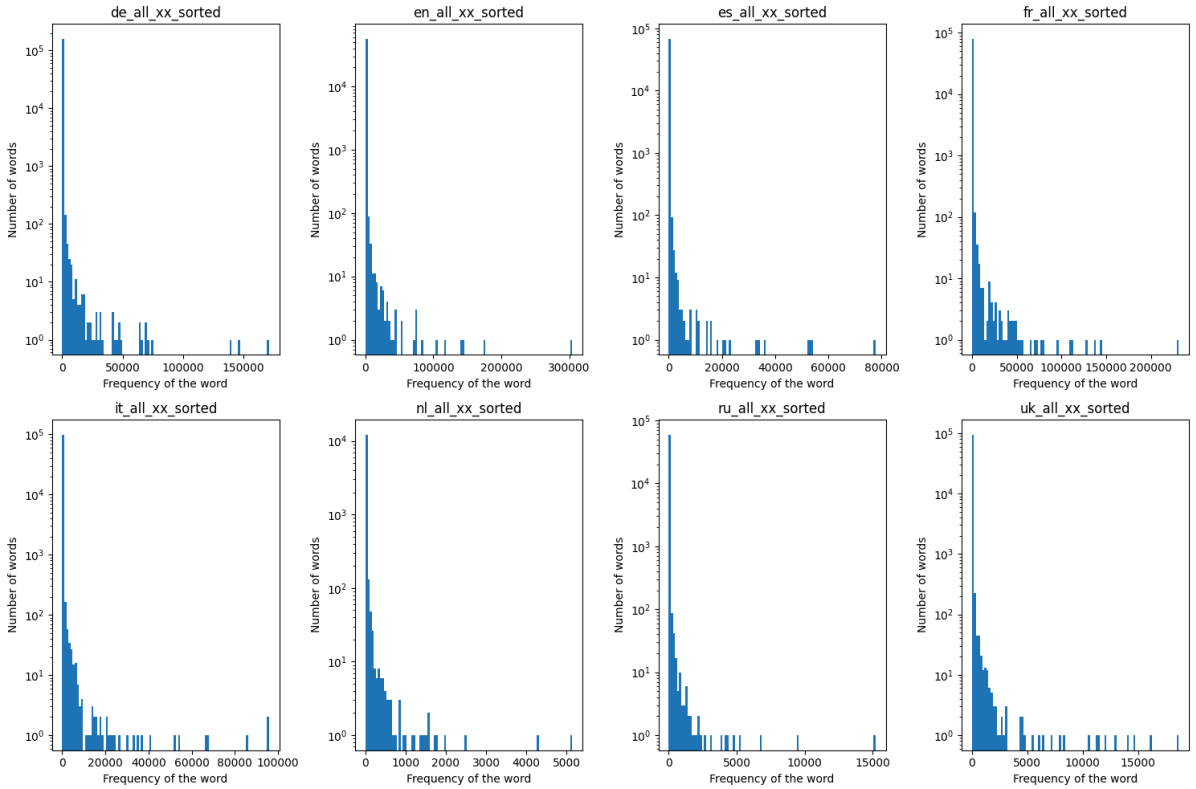
a.) Worthäufigkeit

Die Worthäufigkeit der einzelnen Sprachen, ohne Stoppwörter, wird in folgendem Diagramm abgebildet:



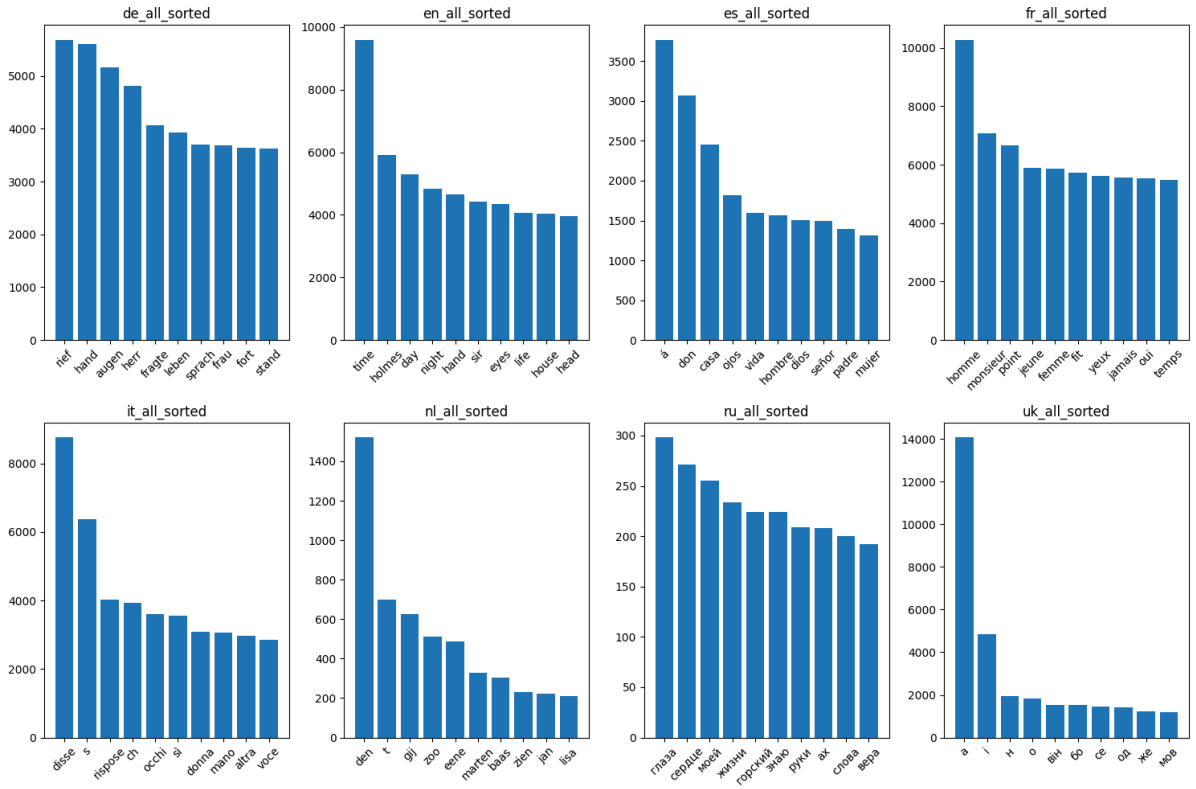
Die X-Achse stellt die Worthäufigkeit dar und zeigt, wie oft die Wörter im Datensatz vorkommen. Auf der Y-Achse wird die Anzahl der einzelnen Wörter angezeigt, die in jeder Häufigkeitsstufe vorkommen. Hohe Spitzen zeigen häufig vorkommende Wörter an. Ausreißer auf der X-Achse könnten möglicherweise nicht entfernte Stoppwörter darstellen.

Ohne die Stoppwörter zu entfernen sieht die Worthäufigkeit wie folgt aus:



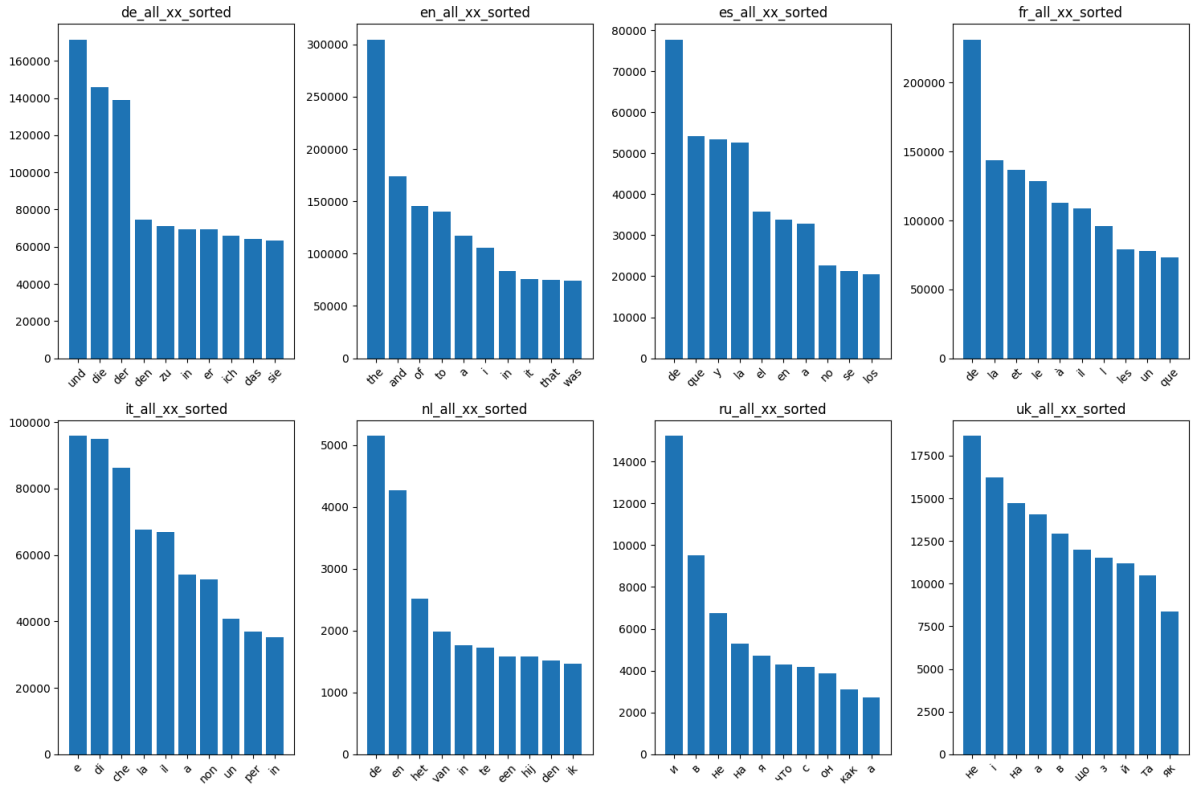
b.) TOP-10 Liste

Die TOP-10 der einzelnen Sprachen, ohne Stoppwörter, werden in folgendem Diagramm abgebildet:



Es ist erkennbar, dass nicht alle Stoppwörter aus den ukrainischen Texten erkannt und entfernt wurden. Im Falle des „a“ könnte dies damit zusammenhängen, dass das kyrillische „а“ nicht dasselbe „a“ ist, wie es im lateinischen Alphabet vorkommt. Im ukrainischen Datensatz kommen sowohl „а“-kyrillisch als auch „a“-lateinisch vor, während die Stoppwortliste für die Ukraine nur das lateinische „a“ enthält.

Ohne die Stoppwörter zu entfernen sieht die TOP-10-Liste wie folgt aus:



Fazit

Hadoop als Framework übernimmt den Großteil des Boilerplate-Codes bei MapReduce-Aufgaben, weist jedoch eine gewisse Lernkurve auf. Nach einer angemessenen Einarbeitungszeit sind die einzelnen Komponenten klar strukturiert und logisch aufgebaut. Es ist möglich, MapReduce-Aufgaben mit relativ geringem Aufwand zu bewältigen.

GitHub-Repository

Der Code und Datensatz für dieses Projekt ist unter https://github.com/FlorianSymmank/Hadoop_WordCount/tree/main einsehbar.