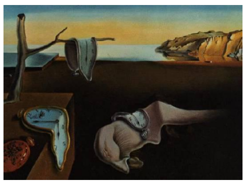

Journal of Statistical Software

November 2012, Volume 51, Issue 5. http://www.jstatsoft.org/

A tm Plug-In for Distributed Text Mining in R

Stefan Theußl
WU Wirtschafts-
universität Wien

Ingo Feinerer
Technische Universität
Wien

Kurt Hornik
WU Wirtschafts-
universität Wien

Abstract

R has gained explicit text mining support with the **tm** package enabling statisticians to answer many interesting research questions via statistical analysis or modeling of (text) corpora. However, we typically face two challenges when analyzing large corpora: (1) the amount of data to be processed in a single machine is usually limited by the available main memory (i.e., RAM), and (2) the more data to be analyzed the higher the need for efficient procedures for calculating valuable results. Fortunately, adequate programming models like MapReduce facilitate parallelization of text mining tasks and allow for processing data sets beyond what would fit into memory by using a distributed file system possibly spanning over several machines, e.g., in a cluster of workstations. In this paper we present a plug-in package to **tm** called **tm.plugin.dc** implementing a distributed corpus class which can take advantage of the **Hadoop** MapReduce library for large scale text mining tasks. We show on the basis of an application in culturomics that we can efficiently handle data sets of significant size.

Keywords: text mining, MapReduce, distributed computing, **Hadoop**.

1. Introduction

In the information age statisticians are confronted with an ever increasing amount of data stored electronically (Gantz *et al.* 2008). This is in particular true for the most natural form of storing information, namely text. Many interesting research questions can be answered via statistical analysis or modeling of huge text corpora. For example news agencies like Reuters provide access to databases of news text documents annotated with rich semantic metadata. These can be employed, e.g., to check daily sentiment in newspaper articles in order to measure interactions between the media and the stock market (Tetlock 2007). E-print repositories such as the arXiv (`http://arXiv.org/`) or the CiteSeerX project (`http://citeseerx.ist.psu.edu/`) allow for harvesting metadata and open access to the corresponding content (i.e., download of full text articles). This information can be used for bibliometric and scientometric analyses, e.g., to find patterns like the formation or development of author or topic (Blei and Lafferty 2007; Griffiths and Steyvers 2004) networks. Furthermore, access to documents written in several centuries, such as the books made available in project Gutenberg (`http://www.gutenberg.org/`), allows one to study how linguistic patterns develop over time. In a recent publication in Science, Michel *et al.* (2011) use 15% of the digitized Google books content (4% of all books ever printed) to study the diffusion of regular English verbs and to probe the impact of censorship on a person's cultural influence over time. This led to the advent of a new research field called *Culturomics*, the application of high-throughput data collection and analysis to the study of human culture.

R, an environment for statistical computing and graphics (R Development Core Team 2012), has gained explicit text mining support via the **tm** package (Feinerer 2012) originally presented in Feinerer, Hornik, and Meyer (2008). This infrastructure package provides sophisticated methods for document handling, transformations, filters, and data export (such as constructing document-term matrices). With a focus on extensibility based on generic functions and object-oriented inheritance, **tm** makes it possible to apply a multitude of existing methods in the R world to text data structures as well.

However, the endeavor to analyze huge text corpora using **tm** is the source of two challenges: (1) the amount of data to be processed in a single machine is usually limited by the available main memory (i.e., RAM), and (2) an increase of the amount of data to be analyzed leads to higher demand for efficient procedures for calculating valuable results. Thus, it is highly imperative to find a solution which overcomes the memory limitation (e.g., by splitting the data into several pieces) and to markedly reduce the runtime by distributing the workload across available computing resources (such as CPU cores or virtual machine instances). Typically, we consider distributed memory platforms like clusters of workstations for such applications since they are scalable in terms of CPUs and memory (disk space and RAM) employed. Furthermore, many different programming models and libraries like the message passing interface (MPI) facilitate working with this kind of *high performance computing* systems. Many of those libraries can directly be employed in R (see Schmidberger, Morgan, Eddelbuettel, Yu, Tierney, and Mansmann 2009, for further references). Still, one open question remains: Is there an efficient way to handle large corpora using R?

In this paper we show that by using the MapReduce distributed programming model (Dean and Ghemawat 2008), and a corresponding implementation called **Hadoop** (The Apache Software Foundation 2010), we are able to transparently distribute the documents on one or several storage entities, apply functions on the subsetted corpus possibly in parallel, and gather results on a cluster of workstations or other (distributed) computing platforms. Typical tasks in text mining like preprocessing can easily be run as parallel distributed tasks without knowing details about the underlying infrastructure. The corresponding extensions to make **tm** recognize such a distributed programming model are encapsulated in a separate plug-in package called **tm.plugin.dc** (Theußl and Feinerer 2012c) offering a seamless integration building on functionality provided by interfaces to MapReduce environments.

The remainder of this paper is organized as follows. In Section 2 we review the typical workflow using the **tm** package and indicate challenges which need to be tackled when working with large data sets. Section 3 summarizes the key concepts of the MapReduce programming model and how it is usually applied in a distributed computing context. The design and implementation of the **tm.plugin.dc** package is discussed in Section 4. The package provides **tm** with supplemental classes and methods in order to benefit from the MapReduce distributed programming model. Additionally, we show how distributed storage can be utilized to facilitate parallel processing of corpora. In Section 5 we present the results of a benchmarking experiment of typical tasks in text mining showing the actual impact on performance in terms of execution time. Section 6 gives an application in culturomics, employing a corpus of several gigabytes of articles from a prominent newspaper to analyze how word usage has changed over 20 years. Section 7 provides computational details. Finally we conclude this paper in Section 8, pointing out directions for future research.

2. The tm package

The text mining infrastructure package **tm** has now become the de facto standard for running text mining applications in R since it provides a transparent way to prepare textual data for statistical analysis and offers easy extensibility via well documented interfaces. In this section we summarize the design (data structures), main features, and important interfaces for extending **tm**. Furthermore, we identify challenges to the standard workflow.

2.1. Data structures and process flow

In **tm** the main data structure is a *corpus*, an entity similar to a database holding text documents in a generic way. It can be seen as a container to store a collection of text documents where additional metadata is provided on both the corpus (e.g., date of creation, creator, etc.) and document level (e.g., annotations, authors, language, etc.). So-called *sources* are used to abstract document acquisitions, e.g., files from a hard disk, over the Internet, or by other connection mechanisms. A separate *reader* function specifies how to actually parse each item delivered by the source (like XML or HTML documents). The latter eases the usage of heterogeneous text formats. For example assume we have a collection of text documents stored in a directory on a local hard disk. We can simply use a predefined source like `DirSource()` which delivers its content. For some news stories from Reuters (see Section 5.1 for a detailed description) in XML format stored in the directory '`Data/reuters`' we can construct the corpus in R via

```
R> library("tm")
R> corpus <- Corpus(DirSource("Data/reuters"),
+    list(reader = readReut21578XML))
```

The function `readReut21578XML()` extracts the actual text content and meta information from the XML document.

Alongside the data infrastructure for acquiring text documents the framework provides tools and algorithms to efficiently work with the documents. Several methods have been implemented to abstract the process of document manipulation. For illustration purposes the **tm** package includes a sample data set containing 50 documents of the Reuters corpus on the topic "Acquisitions" (`acq`). We will use it for demonstration (in particular the sixth document) as it provides easy reproducibility. It can be loaded via

```
R> data("acq")
R> acq[[6]]
```

```
A group of affiliated New York
investment firms said they lowered their stake in Cyclops Corp
to 260,500 shares, or 6.4 pct of the total outstanding common
stock, from 370,500 shares, or 9.2 pct.
    In a filing with the Securities and Exchange Commission,
the group, led by Mutual Shares Corp, said it sold 110,000
Cyclops common shares on Feb 17 and 19 for 10.0 mln dlrs.
 Reuter
```

Typical preprocessing tasks (i.e., data preparation and cleaning) like whitespace removal, stemming or stop word deletion can be applied to individual documents contained in the corpus without difficulty.

Stemming denotes the process of deleting word suffixes to retrieve their radicals, i.e., a word stem (also known as root in linguistics). It typically reduces the complexity without any severe loss of information. One of the best known stemming algorithm goes back to Porter (1980) describing an algorithm that removes common morphological and inflectional endings from English words. The **tm** function `stemDocument()` provides an interface to the Porter stemming algorithm.

```
R> stemmed <- stemDocument(acq[[6]])
R> stemmed

A group of affili New York
invest firm said they lower their stake in Cyclop Corp
to 260,500 shares, or 6.4 pct of the total outstand common
stock, from 370,500 shares, or 9.2 pct.
    In a file with the Secur and Exchang Commission,
th group, led by Mutual Share Corp, said it sold 110,000
Cyclop common share on Feb 17 and 19 for 10.0 mln dlrs.
 Reuter
```

Stop words are words that are so common in a language that their information value is almost zero, i.e., they do not carry significant information (van Rijsbergen 1979). Therefore it is a common procedure to remove such stop words. Similarly to stemming, this functionality is already provided by **tm** via the `removeWords()` function. Removal of whitespace (blanks, tabulators, etc.) and removal of punctuation marks (dot, comma, etc.) can be done via the `stripWhitespace()` and `removePunctuation()` functions.

We denote functions modifying the content of individual text documents in a corpus as *transformations*. Another important concept is *filtering* which basically involves applying predicate functions on collections to extract patterns of interest. See Feinerer *et al.* (2008) for more information about transformations and filtering.

Transformations like whitespace removal, stemming or stop word deletion can easily be applied to all documents contained in a corpus using the `tm_map()` function. The following single call suffices to remove all English stop words from the text corpus `acq`.

```
R> removed <- tm_map(acq, removeWords, stopwords("english"))
R> removed[[6]]
```

```
A   affiliated New York
investment firms   lowered  stake  Cyclops Corp
 260,500 shares,  6.4 pct   total outstanding common
stock,  370,500 shares,  9.2 pct.
    In  filing   Securities  Exchange Commission,
 , led  Mutual Shares Corp,    sold 110,000
Cyclops common shares  Feb 17  19  10.0 mln dlrs.
 Reuter
```

A very common approach in text mining is to break texts into smaller pieces called *tokens*, and use a suitably normalized subset of these as the *terms* representing the text for subsequent computations (see e.g., Section 2.2 in Manning, Raghavan, and Schütze 2008). Such terms are not necessarily words in the sense of Miller (1995), which are strings made from letters in an alphabet.

Package **tm** supports the construction of a so-called *document-term matrix* (DTM) holding frequencies of distinct terms, i.e., the *term frequency* (TF) for each document. When using **tm** DTMs are stored using a simple sparse representation implemented in package **slam** (Hornik, Meyer, and Buchta 2012). DTM construction typically involves preprocessing and counting TFs for each document. The function `DocumentTermMatrix()` is used to *export* such a matrix from a given corpus (the first argument) applying certain preprocessing steps specified via a list of `control` options (the second argument) that should be executed before counting TFs in each document.

```
R> dtm <- DocumentTermMatrix(acq,
+    list(removePunctuation = TRUE, stemming = TRUE))
R> dtm

A document-term matrix (50 documents, 1346 terms)

Non-/sparse entries: 3420/63880
Sparsity           : 95%
Maximal term length: 240
Weighting          : term frequency (tf)
```

Note that the order of transformations to be applied on the corpus can play an important role in terms of run time and actual results. Here we first remove punctuation marks and stem the document before counting individual terms.

The obtained DTMs can be manipulated using the functionality provided by package **slam**, in particular using functions such as `col_sums()`, `row_means()`, or `rollup()` for efficient aggregation. For example we can find the terms which occur most often in the corpus as follows.

```
R> library("slam")
R> head(sort(col_sums(dtm), decreasing = TRUE))

  the  said   and  dlrs   for share 
  394   186   169    97    90    86 
```

2.2. Interfaces

Conceptually we want a corpus to support a set of intuitive operations, like accessing each document in a direct way, displaying and pretty printing the corpus and each individual document, obtaining information about basic properties (such as the number of documents in the corpus), or applying some operation on a range of documents. These requirements are formalized via a set of interfaces which must be implemented by a concrete corpus class:

Subset The `[[` operator must be implemented so that individual documents can be extracted from a corpus.

Display The `print()` and `summary()` methods must convert documents to a format so that R can display them. Additional meta information can be shown via `summary()`.

Length The `length()` method must return the number of documents in the corpus.

Iteration The `tm_map()` function which can be conceptually seen as an `lapply()` has to implement functionality to iterate over a range of documents and applying a given function.

Export For subsequent text analysis it is crucial to define export mechanisms like document-term matrix construction. This allows for using tools other than those provided with **tm**.

The implementation in R via the **tm** package was mainly driven by these conceptual requirements and interface definitions, and is characterized by a virtual corpus class which defines the above set of pre-defined interfaces. Derived corpus classes must implement these in order to support the full range of desired properties. The main advantage of using a virtual class with well-defined interfaces is that instantiated subclasses work with any function aware of the abstract interface definitions but the underlying implementation and representation of internal data structures is completely abstracted.

2.3. Challenges

We identified two challenges when using the **tm** framework. Firstly, big data sets, i.e., data sets which do not fit into main memory like corpora with several millions of documents, cannot easily be constructed and thus processed with the basic facilities provided by **tm**. Secondly, iterations over several millions of documents are rather time consuming. For example performing typical preprocessing steps like stemming or stop word removal on raw text documents can become quite expensive in terms of computing time when the corpus is very large.

Fortunately, operations such as applying transformations and filters are highly amenable to parallelization by construction, as they can separately be applied to each document without side effects. Furthermore, as described in Section 2.1 *sources* are used to abstract document acquisitions. Although we use different sophisticated mechanisms for corpus construction like using database back ends it is conceptually appealing and possible to allocate the storage in a distributed manner since communication is usually not limited by a single bottleneck. Ideally, even subsets of the original data set (the corpus) are stored physically distributed on several machines (e.g., in a cluster of workstations). This will not only allow us to increase storage space for data (scaling with the number of participating machines) but also reduce communication costs for parallel computation since only those documents stored locally on a given machine are to be processed on the respective system. Thus, we can use these two approaches (parallel processing, distributing data) to tackle the challenges indicated above.

Both requirements are often fulfilled by well-established distributed programming models such as MapReduce (Dean and Ghemawat 2008). Typically, MapReduce is used in combination with another important building block: the distributed file system (DFS, Ghemawat, Gobioff, and Leung 2003). This approach readily enables and takes care of data distribution and suitable parallel processing of parts of the data in a functional programming style (Lämmel 2007). Given that the MapReduce model fits to the workflow presented above and corresponding open source software libraries are available, it seems an excellent choice when we need to process large corpora in text mining scenarios.

3. Distributed computing using MapReduce

In this section we describe the programming model and the corresponding implementation (i.e., the underlying software framework) we employed to achieve parallel text mining in a distributed context. MapReduce is a software framework/library originally proposed by Google for large scale processing of data sets. It consists of two important primitives related to concepts from functional programming, namely a `map` function and a `reduce` function. Basically, the `map` function processes a given set of input data (e.g., collections of text files, log files, web sites, etc.) to generate a set of intermediate data which may/is to be aggregated by the `reduce` function.

Note however, that as pointed out in Lämmel (2007) `map` and `reduce` operations in the MapReduce programming model do not necessarily follow the definition from functional programming. It rather aims to support computation (i.e., map and reduction operations) on large data sets on clusters of workstations in a distributed manner. Provided each mapping operation is independent of the others, all maps can be performed in parallel. Indeed, we can express many tasks in text mining in this model, e.g., preprocessing tasks like stemming.

3.1. Programming model

Usually, in this model we consider a set of workstations (nodes) connected by a communication network. Given a set of input data we want to employ these nodes for parallel processing of suitable subsets of the input. *Data locality* is exploited by distributing the data in such a way that parts of the data can efficiently be processed locally on the nodes by individual `map` and aggregated by `reduce` operations. Figure 1 shows this conceptual flow of `map`/`reduce` operations on a given data set.

The `map` function usually transforms its input data into a list of key/value pairs. The MapReduce library takes care of reading the corresponding subset of the data located on each node (local data) from disk and handling the results retrieved from the `map` operation (intermediate data). Basically we have three choices to handle intermediate data. First, we can write the processed data to disk. Second, we can apply another `map` function to the intermediate data and/or third, we can apply the `reduce` function which takes the resulting list of key/value pairs and typically aggregates this list based on the keys. This aggregation results in a single key/value pair for each key. All of these operations can be parallelized over the nodes. Typi-

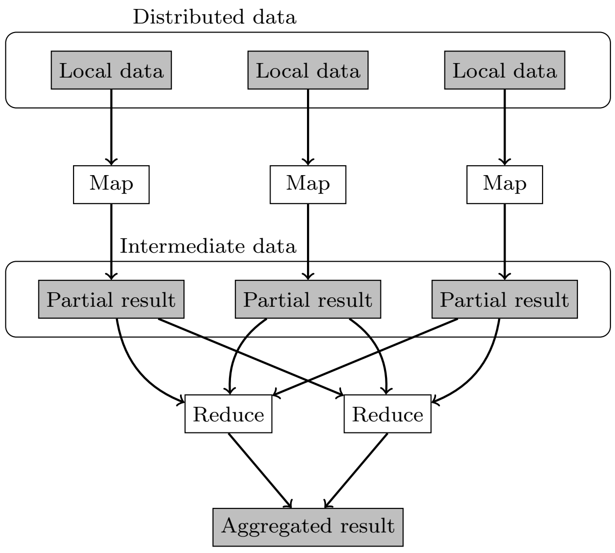

Figure 1: Conceptual flow of data processing when using `map` and `reduce` operations. Each node represents a workstation where operations (white boxes) are applied to local parts of data (gray boxes).

cally, every node applies the same `map` function (the mapper) on its local set of data and some of these nodes aggregate different partial result sets (intermediate data) based on the same `reduce` function (the reducer). However, whereas a mapper typically operates on a single data subset, a reducer may aggregate several partial result sets. All of these operations can be chained.

The major advantage of this so-called data parallelism is that if implemented well this approach theoretically scales over any number of nodes. Furthermore, assigning more than one `map` task to each node advances load balancing since the workload generated by all MapReduce operations is not only distributed across multiple computers or a computer cluster but also as data subsets (which typically correspond to the total number of `map` operations) which can be easily relocated to other nodes by the MapReduce library to avoid overload.

To apply the MapReduce programming model in a distributed text mining context we rely on the open source implementation **Hadoop**.

3.2. Distributed file system

Implementations of MapReduce are typically coupled to a distributed file system (DFS) which assists in data distribution and enables fault tolerance. For example Google offers the Google File System (GFS) to store data as well as intermediate and final results in a distributed manner. Such a file system assists in exploiting data locality which makes parallel execution of tasks faster as additional communication overhead is avoided. Only pieces of data local to each node are considered for processing which improves overall I/O bandwidth. Furthermore, data is replicated on multiple nodes so that failures like crashed nodes or network routing problems will not affect computation. This enables automatic recovery of data in case of system failures. Such a fault tolerant environment is well suited for large clusters of commodity machines – the prime platform for many scientific institutions and companies because of its cost-effectiveness (Barroso, Dean, and Hölzle 2003).

Similar to GFS the **Hadoop** Distributed File System (HDFS, Borthakur 2010) provides such a scalable and fault tolerant architecture. Copying data to the HDFS implies that the given input data is split into parts (physical blocks or separate files). These parts – often called *chunks* – are distributed over the system and replicated over a predefined number of machines. Files are organized hierarchically in directories and identified by path names.

3.3. Software packages

In addition to the MapReduce implementation for distributed manipulation of large data sets and the HDFS, the **Hadoop** framework also includes a utility called ***Hadoop*** *Streaming* implementing a simple interface to **Hadoop** allowing for the usage of MapReduce with any executable or script as the mapper and the reducer. It transforms input data stored on the HDFS and aggregates the results based on the provided scripts.

Several R packages offer functionality based on the MapReduce programming model and/or interface **Hadoop** (Streaming). The package **HadoopStreaming** (Rosenberg 2012) assists in writing proper MapReduce scripts in order to operate on data in a streaming fashion. Similar functionality is offered by the package **mapReduce** (Brown 2012) closely following the framework and nomenclature proposed by Dean and Ghemawat (2008). However, neither package provides facilities for directly interacting with MapReduce libraries. Package **hive** (Theußl and Feinerer 2012b) allows the creation of executable R scripts (Rscript, R Development Core Team 2012) from provided R functions and automatically run them on a **Hadoop** cluster via **Hadoop** Streaming. This approach offers high level access from within R to the **Hadoop** MapReduce and file system functionality. See Appendices A and B for installing and configuring **Hadoop** and examples on how to use **hive**.

Recently the RHadoop project announced the availability of packages **rmr2** (Revolution Analytics 2012b) and **rhdfs** (Revolution Analytics 2012a). These packages provide a **Hadoop** Streaming connector for running MapReduce tasks in R and functions to access the HDFS. However, they have not been made available in one of the standard R repositories thus far.

4. Design and implementation

The bridge between MapReduce libraries and the **tm** infrastructure is characterized by two main design concepts: *distributed storage* and *parallel computation*. The one side of the bridge is designed in such a way that it provides a corpus implementation of the abstract interfaces as outlined in Section 2.2. Its classes and methods can transparently be used in combination with the existing **tm** infrastructure. Via the other side of the bridge we can access and modify data stored on the **Hadoop** Distributed File System (HDFS). Data is processed efficiently in parallel using the **Hadoop** Streaming utility.

In this section we discuss the design of the package **tm.plugin.dc** (Theußl and Feinerer 2012c) building on functionality provided by interfaces to **tm** and to MapReduce environments. We show that by selecting appropriate building blocks we are not only able to employ the tools provided by **Hadoop** but also any abstract registered (distributed) storage and parallel computing environment. Thus, the implementation realized in **tm.plugin.dc** is driven by *Distributed Storage and Lists* as provided with package **DSL** (Theußl and Feinerer 2012a) since it implements both concepts and meets the interface requirements. We use this approach and introduce a new distributed corpus class along with corresponding methods which allow us to analyze large corpora seamlessly without knowing how to use the underlying components of MapReduce or other libraries.

4.1. Distributed storage

Typically, a corpus in **tm** is built by constructing an appropriate data structure holding a sequence of single text documents enriched with metadata which further describes textual content and thus offers valuable insights into the document structure. The content for each text document is acquired via source access and copied into main memory. When applied to large corpora computations may slow down significantly due to high RAM usage as there is a practical limit on the maximum corpus size (by the physical memory size minus overhead by the operating system and other applications).

As pointed out in Section 2.3 we can use MapReduce libraries like **Hadoop** to tackle the challenges which come upon us when dealing with large data sets. Thus, in order to use **Hadoop** Streaming from the **Hadoop** runtime environment, we need the corresponding data to be written to the HDFS. We do this in form of key/value pairs generated for each element in the corpus (i.e., for the individual text documents) as required by the MapReduce paradigm. More generally, a corpus using such a (distributed) storage backend can easily be described as a *list* of key/value pairs written to files with the document ID as the key and the corresponding content (a serialized R object) as the value, respectively. We refer to these collections of key/value pairs contained in files as chunks. In R one must only store unique pointers identifying the individual chunks containing the serialized documents.

Both concepts, distributed storage and lists, are implemented in package **DSL**. The S3 class '`DStorage`' defines a virtual storage where files are kept on a file system which possibly spans over several workstations. Typically, data is distributed automatically among these nodes when using such a file system. Objects of class '`DStorage`' "know" how to use the corresponding file system by supplied accessor and modifier methods. The following file system types are supported:

`"LFS"`: the local file system. This type uses functions and methods from the packages **base** and **utils** delivered with the R base distribution to handle files.

`"HDFS"`: the **Hadoop** Distributed File System. Functions and methods from the package **hive** (Theußl and Feinerer 2012b) are used to interact with the HDFS.

This abstract storage class is mainly used for storing key/value pairs as described above. For efficiency reasons several key/value pairs are put line by line into files of a certain maximum size. Indeed, frameworks like **Hadoop** benefit from such a setup (see Section *Data Organization* in Borthakur 2010). Moreover, package **DSL** allows one to store a set of so-called *revisions* for each operation on objects stored in '`DStorage`'. This enables extremely fast switching between various snapshots of the same objects (like a history with rollback feature known from database systems). Using the term "revision" is mainly motivated by the **Subversion** (SVN, Pilato, Collins-Sussman, and Fitzpatrick 2004) revision concept.

In order to construct a distributed storage object in R the `DStorage()` function from package **DSL** is used. This constructor takes the following arguments:

`type`: The file system type.

`base_dir`: The directory under which chunks containing key/value pairs are to be stored.

`chunksize`: The maximal size of a single chunk.

`keep`: Specifying whether to keep data of all stages in a processing chain as revisions.

In the following example we instantiate a distributed storage of `type "HDFS"` using the system-wide or a user-defined *temporary directory* as the base directory (`base_dir`), a chunk size of 10MB and revisions disabled.

```
R> library("DSL")
R> ds <- DStorage(type = "HDFS", base_dir = tempdir(),
+    chunksize = 10 * 1024^2, keep = FALSE)
R> ds

DStorage.
- Type: HDFS
- Base directory on storage: /tmp/RtmpWOZwMJ
- Current chunk size [bytes]: 10485760
```

Distributed lists are defined by the S3 class '`DList`'. Objects of this class behave similar to standard R lists but use a distributed storage of class '`DStorage`' to store their elements. Distributed lists can easily be constructed using the function `DList()` or be obtained via coercion using the generic function `as.DList()`. Available methods support coercion of R lists and character vectors representing paths to data repositories as well as coercion of '`DList`' objects to lists.

```
R> dl <- DList(letters = letters, numbers = 0:9)
R> dl

A DList with 2 elements

R> l <- as.list(letters)
R> names(l) <- LETTERS
R> dl <- as.DList(l)
R> identical(as.list(dl), l)

[1] TRUE
```

The above example uses the default storage type, namely `"LFS"` for storing list elements. In order to set a user defined storage the `DStorage` argument to the `DList()` constructor is used.

```
R> dl <- DList(letters = letters, numbers = 0:9, DStorage = ds)
```

Furthermore, we can replace the storage assigned to a distributed list. The data is automatically copied to the new storage.

```
R> dl <- DList(letters = letters, numbers = 0:9)
R> DL_storage(dl)

DStorage.
- Type: LFS
- Base directory on storage: /tmp/RtmpWOZwMJ
- Current chunk size [bytes]: 10485760

R> DL_storage(dl) <- ds
R> DL_storage(dl)

DStorage.
- Type: HDFS
- Base directory on storage: /tmp/RtmpWOZwMJ
- Current chunk size [bytes]: 10485760
```

4.2. Parallel computation

Once we have documents stored on the distributed storage as distributed lists, we want to perform computations on the data pieces local to each processing node. Such computations are highly parallel and scale with the number of available workstations. A recurrent function when computing on lists in R is `lapply()` and variants thereof. Conceptually, this is similar to a `map` function from functional programming where a given (R) function is applied to each element of a vector (or in this case a list). Other typical operations on distributed data are *collective* operations. Functions of this type commonly gather or aggregate data based on a given set of instructions. In functional programming the latter is called `reduce` but possible variations also exists in other areas (e.g., in the MPI standard, see Message Passing Interface Forum 1994, 2003).

Package **DSL** offers the following high-level collective and apply-style functions:

`DGather()`: This collective operation is inspired by MPI_GATHER defined in Message Passing Interface Forum (2003). However, instead of collecting results from processes running in parallel, `DGather()` collects the contents of chunks holding the elements of a '`DList`'. By default a named list of length the number of chunks is to be returned. Its elements are character vectors of values from key/value pairs stored in chunks read line by line from the corresponding chunk. Alternatively, `DGather()` can be used to retrieve the keys only.

`DLapply()`: Is an (l)apply-type function which is used to iteratively *apply* a function to a set of input values. In case of `DLapply()` input values are elements of '`DList`' objects (i.e., the value of a key/value pair). A distributed list of the same length is to be returned.

`DMap()`: Is the more general variant of `DLapply()` above where both the key and the value from a key/value pair are taken as input. Thus, keys can also be modified. Moreover, `DMap()` may return an object whose length differs to the original as opposed to `DLapply()`.

`DReduce()`: This collective operation takes a set of (intermediate) key/value pairs and combines values with the same associated key using a given directive (the `reduce` function). By default values are concatenated using the `c()` operator.

Parallel execution of the above operations on a '`DList`' object is ensured by the parallel environment which is associated with the assigned distributed storage. In order to take advantage of the MapReduce parallel computing paradigm for operations on '`DList`' the HDFS storage type must be used which has **Hadoop** Streaming associated. Package **DSL** depends on **hive** for rewriting `map` functions on-the-fly to **Hadoop** `map` functions by creating executable R scripts which are sent to the **Hadoop** environment via the **Hadoop** Streaming utility. In detail, every node is assigned a particular chunk located in the HDFS. As indicated in Section 3.2 this approach allows for low communication overhead as each node typically accesses and computes only on the data physically located at its position. Each list element located in the corresponding chunk gets unserialized and the `map` function is applied. The (again serialized) results are stored on the HDFS and the pointers in the '`DList`' are updated to match the corresponding chunks. This approach allows us to use not only single (multi-core) systems but also highly scalable clusters of workstations.

For LFS-based storage types the associated parallel environment is *multicore* as provided by the **parallel** package (R Development Core Team 2012) available as part of R since version 2.14.0 (see the package vignette for details). This simpler technology can be used without extra configuration on most systems to store and process corpora not fitting into main memory. However, it does not scale beyond the boundaries of a single (possibly multi-core) system.

4.3. The distributed corpus class

Since **tm** corpora are basically lists of objects of class '`TextDocument`' enriched with metadata it seems only natural to encapsulate this storage abstraction in distributed lists as provided with package **DSL**. Appropriate methods for '`DList`' objects ensure that the local files delivered by a source instance are transparently loaded into the distributed storage as a list of key/value pairs.

We call a corpus based on objects of class '`DList`' a *distributed corpus*. This new type of corpus implemented in package **tm.plugin.dc** reduces the main memory consumption drastically since even for millions of documents we keep only the pointers to the (serialized) documents in memory, which occupy just a few megabytes. Technically, we implemented the class '`DCorpus`' which inherits from a standard corpus on the one hand and from class '`DList`' on the other hand. Building the bridge in this way allows us to utilize such a corpus instance in all use cases of a standard corpus by directly employing corresponding high-level collective and apply-style functions like `DGather()`, `DMap()` and `DReduce()`. Since the **tm** infrastructure is designed in a very modular and generic way as described in Section 2.2, we only needed to write methods for a few generic functions in order to abstract the underlying distributed storage.

Transformations triggered via `tm_map()` are applied to elements of '`DCorpus`' objects simply by using `DLapply()` from **DSL** instead of `lapply()` defined in the default method in **tm**. The construction of document-term matrices (DTMs) is a combination of `map` and `reduce` steps.

```
R> intermed <- DReduce(DMap(x, map), reduce)
```

In the `map` step preprocessing as described above is applied to the given corpus `x` and the remaining terms are counted and stored so that a term represents the key and the value is a list of document ID and term frequency. The `reduce` step then aggregates (concatenates), for each term, ID and the corresponding term frequency. The result delivered with `DReduce()` is stored as intermediate data (`intermed`). All of these steps may run in parallel. Eventually, based on `intermed` the DTM is constructed from the individual term vectors read from the distributed storage via `DGather()` and combined on the master node.

This implementation allows us to seamlessly use '`DCorpus`' objects in all scenarios supported by the **tm** package and beyond that to hold large amount of textual data as distributed corpora in R.

4.4. Using package tm.plugin.dc

The package **tm.plugin.dc** has to be attached in order to make use of the class '`DCorpus`'.

```
R> library("tm.plugin.dc")
```

In order to take advantage of the MapReduce parallel computing paradigm for operations on '`DCorpus`' we need to specify to use the HDFS storage type which has **Hadoop** Streaming associated. This requires a working **Hadoop** installation (e.g., on a cluster of workstations) and package **hive** installed which is loaded automatically in the background. **hive** offers an interface to file system accessors and high-level access to **Hadoop** Streaming. The function `DStorage()` is used to prepare the corresponding storage to be used for text mining tasks.

```
R> storage <- DStorage(type = "HDFS", base_dir = "/tmp/dc")
```

Similar to the standard process flow presented in Section 2.1 the data has to be retrieved from the specified source via

```
R> dc <- DCorpus(DirSource("Data/reuters"),
+    list(reader = readReut21578XML), storage)
```

or we can *coerce* a standard '`Corpus`' to '`DCorpus`' by using the generic `as.DCorpus()`.

```
R> data("acq")
R> dc <- as.DCorpus(acq, storage)
R> dc

DCorpus. A corpus with 50 text documents
```

Both functions take a pre-defined storage object as argument specifying that data (either delivered by the source or contained in the corpus) is to be stored as chunks on the given (distributed) storage (see Section 4.1). After that, appropriate methods ensure that '`DCorpus`' can be handled as defined for '`Corpus`'. This allows for a seamless integration of the HDFS or any other storage type defined in **tm.plugin.dc** into **tm** without changing the user interface. For example calling `tm_map()` has the same effects as shown in Section 2.1 but uses the **Hadoop** framework for applying the provided `map` functions instead.

```
R> dc <- tm_map(dc, stemDocument)
R> stemmed <- tm_map(acq, stemDocument)
R> all(sapply(seq_along(acq), function(x) identical(dc[[x]], stemmed[[x]])))
```

```
[1] TRUE
```

The processed documents are still stored on the HDFS since by concept the return value of `tm_map()` must be of the same class as the input value. They can easily be retrieved with corresponding accessor methods.

Furthermore, since '`DStorage`' offers to store revisions of contained objects, we are able to switch between various snapshots of the same corpus. This enables methods to work on different levels in a preprocessing chain, e.g., before and after stemming. In addition revisions allow for backtracking to earlier processing states, a concept similar to rollbacks in database management systems. Revisions are enabled by default and can be retrieved or set using the functions `getRevisions()` and `setRevision()`, respectively.

```
R> revs <- getRevisions(dc)
R> revs

[1] "DSL-20120606-162257-fsaxxphaws" "DSL-20120606-162256-yyoqrbeqck"

R> dc <- setRevision(dc, revs[length(revs)])
R> all(sapply(seq_along(acq), function(x) identical(dc[[x]], acq[[x]])))

[1] TRUE
```

The first element in the revision vector always represents the most recent revision (such as the resulting corpus after applying transformations) and the last element represents the revision of the original corpus. Revisions can be turned off using

```
R> keepRevisions(dc) <- FALSE
```

Another method for '`DCorpus`' ensures that the DTM is constructed in parallel. Based on the storage (here HDFS) the **Hadoop** Streaming utility is used to construct both, the term vectors per document (the `map` step) and triplets of the form $(term, ID, tf)$ (the `reduce` step). Finally, after all `map` and `reduce` steps succeeded in their respective task the resulting matrix is constructed from the aggregated data stored in the HDFS on the master node. Note that the **Hadoop** runtime environment allows us to set the number of parallel working reducers. This can be done via the function `hive::hive_set_nreducer()`. Otherwise only one reducer will be used.

```
R> DocumentTermMatrix(dc, list(stemming = TRUE, removePunctuation = TRUE))

A document-term matrix (50 documents, 1464 terms)

Non-/sparse entries: 3502/69698
Sparsity           : 95%
Maximal term length: 249
Weighting          : term frequency (tf)
```

5. Performance

In this section we illustrate the performance of the distributed corpus implementation in two experiments. In the first experiment we study the runtime behavior of typical text mining tasks when using the plug-in package introduced in Section 4.3. We compare the results to the parallel computing approach implemented in the **tm** package. The latter uses the *Message Passing Interface* (MPI, Message Passing Interface Forum 1994, 2003) and corresponding R interface packages for parallel processing.

In the second experiment we investigate the performance of our 'DCorpus' implementation when using corpora showing different characteristics. In particular we are interested in the runtime and throughput behavior when varying corpus size or document size.

5.1. Data

For our performance experiments we consider four corpora: the *Reuters-21578* corpus, a collection of *Research Awards Abstracts* from the National Science Foundation (NSF), the *Reuters Corpus Volume 1* (RCV1) and the *New York Times (NYT) Annotated Corpus*. Table 1 gives an overview on the different corpora by showing the following figures: the number of documents included in the corpus (the length of the corpus), the mean number of characters per document (the mean document length), and the disk space needed to store the corpus (the corpus size).

Pre-built data packages for the freely redistributable Reuters-21578 (**tm.corpus.Reuters21578**) and NSF (**tm.corpus.NSF**) corpora can be downloaded from the data repository of the Institute for Statistics and Mathematics of WU Wirtschaftsuniversität Wien (http://datacube.wu.ac.at/). Packages for the RCV1 and NYT corpora cannot be made publicly available due to license restrictions but can be obtained from the first author if the right to use the data can be verified.

Reuters-21578

The Reuters-21578 data set (Lewis 1997) contains stories collected by the Reuters news agency. The data set is publicly available and has been widely used in text mining research within the last decade. It contains 21,578 short to medium length documents in XML format (obtainable e.g., from http://ronaldo.cs.tcd.ie/esslli07/data/) covering a broad range of topics, like mergers and acquisitions, finance, or politics. To download the corpus use:

```
R> install.packages("tm.corpus.Reuters21578",
+    repos = "http://datacube.wu.ac.at", type = "source")
R> library("tm.corpus.Reuters21578")
R> data("Reuters21578")
```

NSF Research Awards abstracts

This data set consists of 129,000 plain text abstracts describing NSF awards for basic research submitted between 1990 and 2003. The data set can be obtained from the *UCI Machine Learning Repository* (Frank and Asuncion 2010). The corpus is divided into three parts. We used the largest part (*Part 1*) in our experiments.

	No. of docs	Mean no. of char./doc[1]	Corpus size [MB][2]
Reuters-21578	21,578	736.39/834.42	87
NSF Abstracts (Part 1)	51,760	2,895.66	236
RCV1	806,791	1,426.01	3,804
NYT Annotated Corpus	1,855,658	3,303.68/3,347.97	16,160

Table 1: Number of included documents, mean number of characters per document, and uncompressed size on the file system for each corpus. 1: With/without considering empty documents. 2: Calculated with the Unix tool `du`.

```
R> install.packages("tm.corpus.NSF",
+    repos = "http://datacube.wu.ac.at", type = "source")
R> library("tm.corpus.NSF")
R> data("NSF_Part1")
```

Reuters Corpus Volume 1

Lewis, Yang, Rose, and Li (2004) introduced the RCV1 consisting of about 800,000 (XML format) documents as a test collection for text categorization research. The documents contained in this corpus were sent over the Reuters newswire (`http://www2.reuters.com/media/newswires/`) during a 1-year period between 1996-08-20 and 1997-08-19. RCV1 covers a wide range of international topics, including business & finance, lifestyle, politics, sports, etc. The stories were manually categorized into three category sets: topic, industry and region.

NYT annotated corpus

The largest data set in our collection contains over 1.8 million articles published by the New York Times between 1987-01-01 and 2007-06-19 (Sandhaus 2008). Documents and corresponding metadata are provided in an XML like format: News Industry Text Format (NITF).

5.2. Procedure

The first experiment consists of running several preprocessing steps and constructing DTMs which usually constitute the major computation effort. With the help of the new class ‘`DCorpus`’ delivered with the **tm.plugin.dc** package we can transparently use **Hadoop**, or more specifically, the HDFS, as “extended” memory to store corpora. Parallelization of transformations (via `tm_map()`) and DTM construction (via `DocumentTermMatrix()`) is then supported by appropriate methods using the **Hadoop** Streaming utility (see Section 4.3). In this experiment we measure the runtime behavior of individual tasks using a selected number of CPUs in order to demonstrate the performance gain in terms of speedup. Moreover, since **tm** supports parallelization of transformations, filter operations, and DTM construction via MPI we can use the results obtained in this approach to benchmark our ‘`DCorpus`’ implementation.

The MPI approach as implemented in the **tm** infrastructure is mainly motivated by the fact that most operations on the documents are independent from the results of other operations. As a consequence there is plenty of room for parallelization, i.e., parallel execution of code fragments on multiple processors with relatively small overhead. This is especially of interest since hardware performance gains during the last years mainly stem from multi-core or multi-processor systems instead of faster (e.g., higher clock frequency) single core processors. Support can easily be (de-)activated via `tm_startCluster()` and `tm_stopCluster()` calls which start up/stop an MPI environment or use an existing instance if already running. Internally the **snow** (Rossini, Tierney, and Li 2003; Tierney, Rossini, Li, and Sevcikova 2012) package is used to manage an MPI cluster which in turn delegates the parallel execution triggered via the `parLapply()` function to the **Rmpi** package (Yu 2002, 2012).

This approach predates and coincides with the efforts taken with the recent **parallel** package to boost parallel execution in R at a whole. The code stays relatively simple since by design the parallel execution of `lapply()` operations is ensured (e.g., via `parLapply()`). This enables the usage on multi-core systems (via multiple instances on individual cores running on a single physical workstation) and on multiprocessor systems (via accessing multiple physically distributed machines). In any case, `parLapply()` splits up the input corpus into a set of suitable chunks of documents and distributes them via MPI. Then the chunks get processed by the individual participating nodes and results are collected again via MPI.

However, on the master node (running the controlling R instance of an MPI cluster) the whole data set stays in the main memory while parts of the data are being processed in parallel on the worker nodes (running R processes for executing operations defined via `parLapply()`). Thus, in contrast to the MapReduce approach the **tm**/MPI implementation is limited by the main memory of the calling machine in terms of data set size. Even if we are able to run several steps in parallel, the whole corpus has to be loaded into RAM initially. As a consequence we are limited to using the Reuters-21578 and the NSF Research Awards Abstracts (Part 1) corpora.

In the second experiment we export DTMs from all four corpora in order to investigate how runtime and throughput (measured as the number of processed characters or bytes per second) is affected with respect to the different characteristics of the corpora employed. Timings also consider the full preprocessing chain applied before constructing the DTM.

All individual tasks were repeatedly (three times) run for a selected number of CPUs.

5.3. Results

Figure 2 shows the runtime improvements achieved in the first experiment where **tm** uses parallelization via MPI (solid line) and **Hadoop** (dashed line). As test set we used the complete Reuters-21578 data set (upper row) and part 1 of the NSF data set (lower row), and performed stemming (left) and stopword removal (middle) for each document in the corpus. We set the number of processor cores available to a range from one to 32. The figure depicts the averaged runtime (three runs per setting) necessary to complete all operations, showing a clear indication how **tm** profits from multi-core or distributed parallelization of typical preprocessing steps on a realistic data set. Both approaches scale almost linearly with the amount of processing cores. Interestingly, MPI scales superlinearly in a few cases, e.g., for stopword removal on the Reuters-21578 data set. One possible explanation is that `gsub()`, which is internally used to replace stopwords with an empty string, takes significantly more time on longer strings (i.e., its asymptotic runtime behavior is bigger than $O(n)$ where n denotes the input string length).

Preprocessing is an embarrasingly parallel computing task, thus we expect a speedup $S(p) = p$ when employing p processing cores. However, Amdahl's law (Amdahl 1967) states that if only

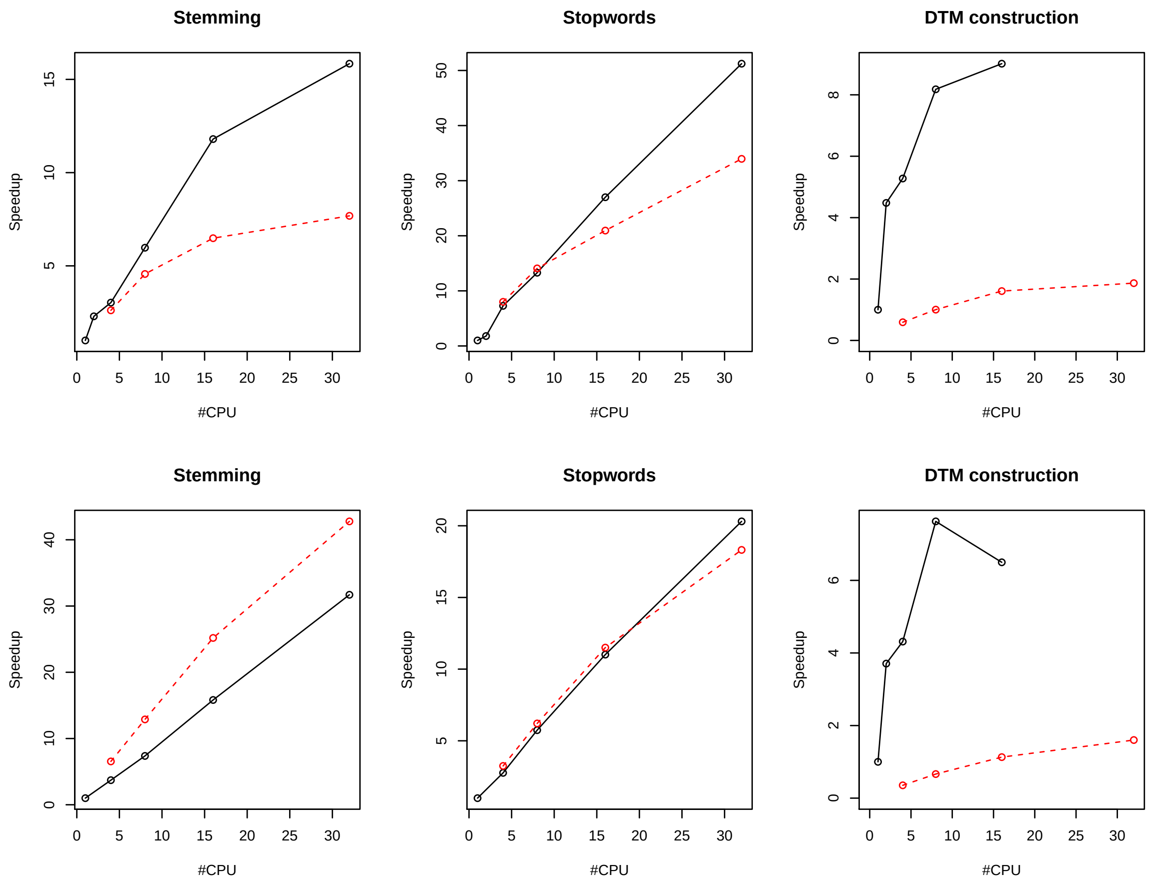

Figure 2: Runtime in seconds for stemming, stopword removal, and DTM construction on the full Reuters-21578 data set (upper row) and on part 1 of the NSF data set (lower row) utilizing either **Hadoop** (dashed line) or MPI (solid line) with up to 32 processing cores.

a fraction f of a given task can be made parallel, the speedup can be calculated as $S(p) = \frac{1}{(1-f)+f/p}$. Since in Figure 2 speedups for **Hadoop** are not linear but for MPI almost are we infer that the fraction $(1-f)$ must be higher for **Hadoop** than for MPI. Thus, costs for starting the **Hadoop** framework (= overhead) must be higher. This is especially relevant for smaller data sets and for computationally cheap operations. In such cases MPI easily outperforms **Hadoop**. For large corpora and computationally expensive operations the **Hadoop** overhead is negligible which makes **Hadoop** the natural choice for large data sets. On clusters of workstations **Hadoop** additionally addresses typical problems like network transfers (limited bandwidth, low latency) and data handling (automatic data split, global access, redundancy) by using a distributed file system. This makes the MapReduce approach the preferred choice for large corpora compared to the MPI approach where many of these problems must be solved manually.

Results for constructing DTMs are shown in Figure 2 on the right for Reuters-21578 and NSF, respectively. Interestingly, MPI outperforms **Hadoop** in this case since communication costs are lower when term vectors are transferred. Nevertheless, the MPI approach seems to be

	Runtime [s]	Throughput [k char/s]	Throughput [MB/s]
Reuters-21578	93	193.6	0.94
NSF Abstracts (Part 1)	291	515.0	0.81
RCV1	5805	198.2	0.66
NYT Annotated Corpus	8330	745.8	1.94

Table 2: Corpus processing statistics. Constructing DTMs with 32 **Hadoop** nodes on a cluster of workstations (see Section 7 for details).

quite unstable since speedups do not always increase when adding a core. Furthermore, we did not manage to get results when using 32 cores probably due to configuration or network setup issues. But most importantly in contrast to **Hadoop** where data is almost always kept on disk, MPI holds data in memory and thus scalability is limited.

The second experiment only employing the **Hadoop** framework reveals that throughput increases significantly for large data sets as seen in Table 2. Here, throughput is measured as the number of 1,000 characters per second ([k char/s]) on the one hand and megabytes per second ([MB/s]) on the other hand. From this table we see some interesting behavior compared to the characteristics of the individual corpora shown in Table 1.

Corpora containing documents with more content (in terms of the mean document length) are processed faster and the mean document length seems to affect the throughput more than the raw corpus size. One possible explanation for the former behavior is that fewer I/O operations have to be performed compared to the total corpus size since we iterate over the documents in the corpus and not over equal sized text chunks. Furthermore, this might also influence the latter behavior as only for the largest data set we observed a significant gain in throughput where load balancing of **Hadoop** is leveraged and becomes effective for the given number of nodes.

To sum up, **Hadoop** is the technology of choice for (1) corpora containing documents with lots of text and/or (2) corpora which are large in terms of disk space required.

6. Application

It was well established in corpus linguistics (e.g., Francis and Kučera 1982) that (word-type) term frequency distributions obtained from large text collections are typically heavily skewed, with relatively few terms covering most of the texts. Are texts maybe getting increasingly "simple" over time, in the sense of increasing coverage by the basic vocabulary? This is a typical question in the spirit of the new exciting field of culturomics (Michel *et al.* 2011), which deals with the development of human behavior and culture reflected in language and word usage.

In this section we add to this field by investigating how the text coverage in newspaper articles has developed over time. Specifically, we analyze the multitude of text documents published by the New York Times between 1987-01-01 and 2007-06-19. The corresponding corpus consists of 1,855,658 short- to medium-length articles from various genres with a mean of 552 terms per document. As such, this corpus is too large for being handled with the standard text mining tool chain available in R. However, we can use the distributed framework presented in this paper in order to process the corpus quite efficiently. This makes it possible to not only

Size	Coverage	Coverage NYT
1000	0.72	0.75
2000	0.80	0.84
3000	0.84	0.88
4000	0.87	0.91
5000	0.89	0.92
6000	0.90	0.93
15851	0.98	0.97

Table 3: Vocabulary coverage in the NYT Corpus compared to standard English text coverage as identified by Francis and Kučera (1982).

investigate the culturomics question at hand, but also to subject the corpus to a variety of other statistical analyses.

First, we compare the text coverage given a fixed vocabulary size for the NYT corpus with the results of Francis and Kučera (1982) (see also `http://en.wikipedia.org/wiki/Vocabulary`). Let $\mathcal{T}$ be a set of different terms, then text coverage is defined as follows.

$$\text{coverage} = \frac{\text{number of terms in a given text exactly matching a term in } \mathcal{T}}{\text{number of all terms in the text}}$$

In sum we find 1,023,484,418 terms in the whole corpus from which we can derive our vocabulary of 547,400 unique terms. This was conveniently achieved by reading all text documents given in the New York Times corpus into a '`DCorpus`' and subsequently deriving the DTM.

Table 3 shows that by knowing the 2,000 English words with the highest frequency, one would know on average 80% of the terms in English texts or 84% of the terms in NYT articles.

However, text coverage using a given vocabulary is not necessarily stable over time since language use may change (see e.g., Hogg and Denison 2008). To investigate how the active vocabulary has changed over time, we can use the *date of publication* metadata contained in the corpus to easily derive a time series of text coverages for a given vocabulary by suitably aggregating the DTM. We know that language texts could simultaneously get "more complicated" in the sense that far tails of the term frequency distribution get heavier, i.e., that increasingly more terms are needed to cover the remaining words. However, we do not pursue the latter issue here as it requires much more extensive natural language processing (NLP) such as named entity recognition and morphological standardization.

Figure 3 shows that text coverage is decreasing almost linearly by roughly 1% over 20 years considering the 1,000 and 4,000 most often used terms in the whole corpus, respectively. Considering this scenario we might conclude that texts are not getting simpler over time.

However, if we consider only stop words, i.e., words that appear in general (in this case English) texts frequently but do not carry significant information, the picture changes. Over the whole NYT corpus we found 332 matching stop words given the list of predefined stop words available in **tm**. The average text coverage using only stop words is 0.54. Figure 4 reveals that text coverage driven by **tm**'s stop words dictionary increases by more than 1% in the first decade until 1997 and remains stable over the second (we do not have a satisfactory explanation for this structural break). This suggests that while texts were getting "more specialized", the amount of non-significant content also increased.

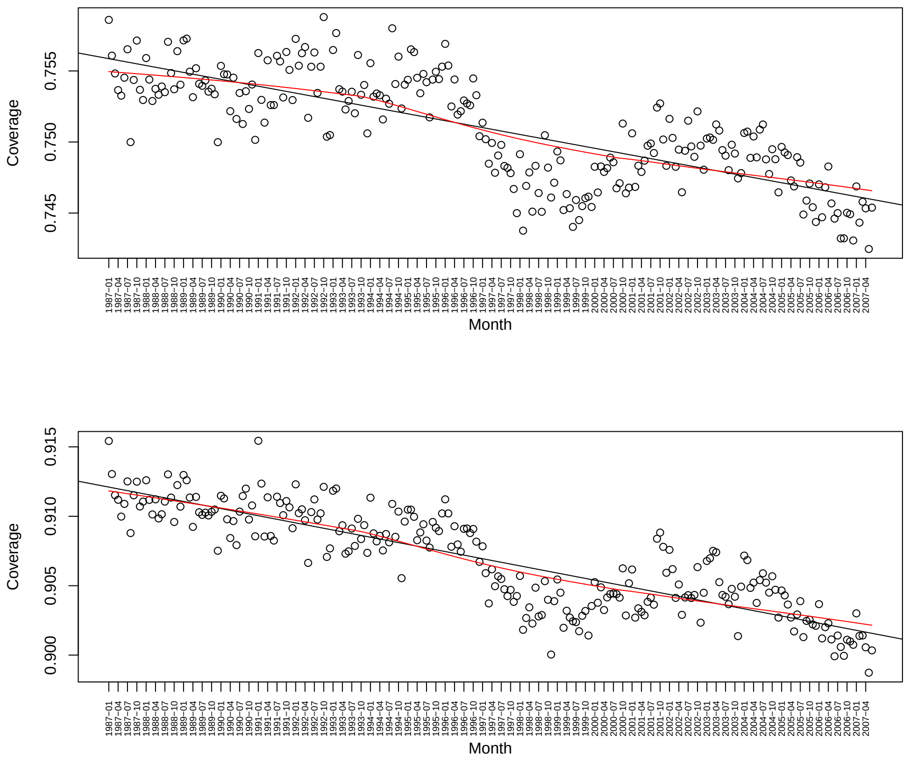

Figure 3: Monthly text coverage using top 1000 (top) and 4000 (bottom) terms in the NYT corpus between 1987-01-01 and 2007-06-19.

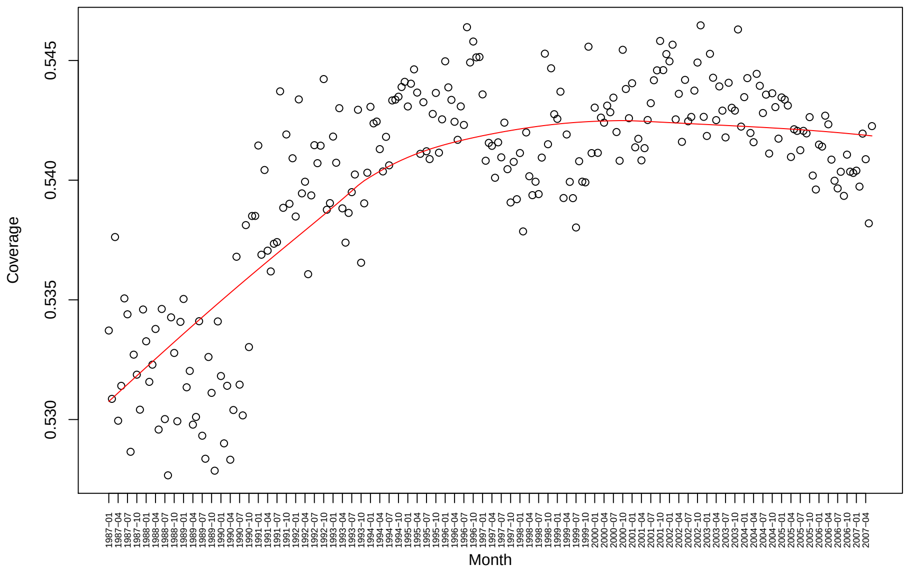

Figure 4: Monthly text coverage using stop words in the NYT corpus.

7. Computational details

All the runtime experiments and applications presented in this paper were performed on *cluster@WU*, the high performance computing cluster environment of WU Wirtschaftsuniversität Wien. Each node of the cluster consists of an Intel Core 2 Duo CPU 2.4 GHz, 110 GB of local (SATA) hard disk storage reserved for the HDFS, and 4 GB of main memory. All nodes are connected by standard Gigabit Ethernet network in a flat network topology. Parallel jobs have been submitted with the Sun Grid Engine version 6.2 update 3. On cluster@WU the parallel environments **Hadoop** (MapReduce) and OpenMPI (MPI) are available in versions 0.20.2 and 1.3.3, respectively. All R code has been executed via version 2.14.0 of R.

8. Conclusion

This paper has introduced an approach applying the distributed programming paradigm MapReduce to advance feasibility and performance of suitable text mining tasks in R. We showed that distributed memory systems can be effectively employed within this model to preprocess large data sets by adding layers to existing text mining infrastructure packages. We also indicated that data parallelism can very easily be achieved using such an integrated framework without altering the handling of current text mining software available in R (i.e., the **tm** package). This is done via the class '`DCorpus`' implemented in package **tm.plugin.dc**. Appropriate methods make use of a distributed storage (such as the **Hadoop** distributed file system) and a corresponding distributed computing framework (MapReduce). A benchmark experiment showed that applying MapReduce in combination with R on text mining tasks is a very promising approach. The results presented in this paper show a significant performance gain over the sequential code as well as very good scalability when employing the distributed memory model. Such an approach enables us to process large data sets like the NYT corpus and to use R's rich statistical functionalty for large scale text mining applications. We showed this in a culturomics application scenario.

With the release of package **tm.plugin.dc** we now have two options for handling text corpora in R: using functionality provided with **tm** where data is kept in main memory or using derived corpus classes where data stays on disk and is only accessed when needed. Unfortunately, no general rule exists that defines the most efficient option for a certain text mining application. Of course, depending on corpus size and available RAM on the main workstation, i.e., when corpora get too big, one has to use the distributed corpus implementation presented in this paper. In other scenarios, i.e., for corpus sizes which are manageable with standard tools, it needs to be investigated which approach is the most promising to be employed. Ideally, **tm** would implement suitable heuristics for choosing the best option given a set of criteria (corpus size, network topology, number of computing nodes, overhead induced by the parallel environment, available RAM, etc.) automatically.

References

Amdahl GM (1967). "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities." In *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pp. 483–485. ACM, New York.

Barroso LA, Dean J, Hölzle U (2003). "Web Search for a Planet: The Google Cluster Architecture." *IEEE Micro*, **23**(2), 22–28.

Blei DM, Lafferty JD (2007). "A Correlated Topic Model of Science." *The Annals of Applied Statistics*, **1**(1), 17–35.

Borthakur D (2010). "HDFS Architecture." *Document on **Hadoop** Wiki.* URL http://hadoop.apache.org/common/docs/r0.20.2/hdfs_design.html.

Brown C (2012). ***mapReduce**: Flexible mapReduce Algorithm for Parallel Computation.* R package version 1.2.6, URL http://CRAN.R-project.org/package=mapReduce.

Dean J, Ghemawat S (2008). "MapReduce: Simplified Data Processing on Large Clusters." *Communications of the ACM*, **51**(1), 107–113.

Feinerer I (2012). ***tm**: Text Mining Package.* R package version 0.5-7.1, URL http://CRAN.R-project.org/package=tm.

Feinerer I, Hornik K, Meyer D (2008). "Text Mining Infrastructure in R." *Journal of Statistical Software*, **25**(5), 1–54. URL http://www.jstatsoft.org/v25/i05/.

Francis WN, Kučera H (1982). *Frequency Analysis of English Usage: Lexicon and Grammar.* Houghton Mifflin.

Frank A, Asuncion A (2010). "UCI Machine Learning Repository." URL http://archive.ics.uci.edu/ml/.

Gantz JF, Chute C, Manfrediz A, Minton S, Reinsel D, Schlichting W, Toncheva A (2008). "The Diverse and Exploding Digital Universe: An Updated Forecast of Worldwide Information Growth through 2011." *IDC White Paper.* URL http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf.

Ghemawat S, Gobioff H, Leung S (2003). "The Google File System." In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pp. 29–43. ACM Press, New York.

Griffiths TL, Steyvers M (2004). "Finding Scientific Topics." *Proceedings of the National Academy of Sciences of the United States of America*, **101**, 5228–5235.

Hogg R, Denison D (eds.) (2008). *A History of the English Language.* Cambridge University Press.

Hornik K, Meyer D, Buchta C (2012). ***slam**: Sparse Lightweight Arrays and Matrices.* R package version 0.1-26, URL http://CRAN.R-project.org/package=slam.

Lämmel R (2007). "Google's MapReduce Programming Model – Revisited." *Science of Computer Programming*, **68**(3), 208–237.

Lewis D (1997). "Reuters-21578 Text Categorization Test Collection." URL http://www.daviddlewis.com/resources/testcollections/reuters21578/.

Lewis DD, Yang Y, Rose TG, Li F (2004). "RCV1: A New Benchmark Collection for Text Categorization Research." *The Journal of Machine Learning Research*, **5**, 361–397.

Manning CD, Raghavan P, Schütze H (2008). *An Introduction to Information Retrieval*, volume 1. Cambridge University Press. URL http://nlp.stanford.edu/IR-book/.

Message Passing Interface Forum (1994). *MPI: A Message-Passing Interface Standard*. URL http://mpi-forum.org/.

Message Passing Interface Forum (2003). *MPI-2: Extensions to the Message-Passing Interface*. URL http://mpi-forum.org/.

Michel JB, Shen YK, Aiden AP, Veres A, Gray MK, The Google Books Team, Pickett JP, Hoiberg D, Clancy D, Norvig P, Orwant J, Pinker S, Nowak MA, Aiden EL (2011). "Quantitative Analysis of Culture Using Millions of Digitized Books." *Science*, **331**, 176–182.

Miller GA (1995). "WordNet: A Lexical Database for English." *Communications of the ACM*, **38**(11), 39–41.

Pilato CM, Collins-Sussman B, Fitzpatrick BW (2004). *Version Control with **Subversion***. O'Reilly. Full book available online at http://svnbook.red-bean.com/.

Porter M (1980). "An Algorithm for Suffix Stripping." *Program*, **3**, 130–137.

R Development Core Team (2012). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

Revolution Analytics (2012a). ***rhdfs**: R and **Hadoop** Distributed Filesystem*. R package version 1.0.5, URL http://github.com/RevolutionAnalytics/RHadoop/wiki/Downloads.

Revolution Analytics (2012b). ***rmr2**: R and **Hadoop** Streaming Connector*. R package version 2.0.1, URL http://github.com/RevolutionAnalytics/RHadoop/wiki/Downloads.

Rosenberg DS (2012). ***HadoopStreaming**: Utilities for Using R Scripts in **Hadoop** Streaming*. R package version 0.2, URL http://CRAN.R-project.org/package=HadoopStreaming.

Rossini A, Tierney L, Li N (2003). "Simple Parallel Statistical Computing in R." *Working Paper 193*, UW Biostatistics Working Paper Series. URL http://www.bepress.com/uwbiostat/paper193/.

Sandhaus E (2008). "The New York Times Annotated Corpus." URL http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2008T19.

Schmidberger M, Morgan M, Eddelbuettel D, Yu H, Tierney L, Mansmann U (2009). "State of the Art in Parallel Computing with R." *Journal of Statistical Software*, **31**(1), 1–27. URL http://www.jstatsoft.org/v31/i01/.

Tetlock P (2007). "Giving Content to Investor Sentiment: The Role of Media in the Stock Market." *The Journal of Finance*, **62**(3), 1139–1168.

The Apache Software Foundation (2010). "**Hadoop**." Release 0.20.2, URL http://hadoop.apache.org/.

Theußl S, Feinerer I (2012a). ***DSL**: Distributed Storage and Lists.* R package version 0.1-2, URL http://CRAN.R-project.org/package=DSL.

Theußl S, Feinerer I (2012b). ***hive**: **Hadoop** InteractiVE.* R package version 0.1-15, URL http://CRAN.R-project.org/package=hive.

Theußl S, Feinerer I (2012c). *Text Mining Distributed Corpus Plug-In.* R package version 0.2-4, URL http://CRAN.R-project.org/package=tm.plugin.dc.

Tierney L, Rossini A, Li N, Sevcikova H (2012). ***snow**: Simple Network of Workstations.* R package version 0.3-10, URL http://CRAN.R-project.org/package=snow.

van Rijsbergen CJ (1979). *Information Retrieval*, volume 2. Butterworths. URL http://www.dcs.gla.ac.uk/Keith/Preface.html.

Yu H (2002). "**Rmpi**: Parallel Statistical Computing in R." *R News*, **2**(2), 10–14. URL http://www.r-project.org/doc/Rnews/.

Yu H (2012). ***Rmpi**: Interface (Wrapper) to MPI (Message-Passing Interface).* R package version 0.6-1, URL http://CRAN.R-project.org/package=Rmpi.

A. Installing Hadoop

In this section we describe how to set up the **Hadoop** framework in (pseudo) distributed operation. This description is based on `http://hadoop.apache.org/common/docs/r0.20.2/quickstart.html` and focuses on the Linux operating system as this platform is suggested on the website for production use. For a more detailed installation instruction and for other platforms than Linux we refer to the above website.

A.1. Basic Hadoop environment

Since `Java` and the command line utility `ssh` are needed for operation both programs have to be installed on all involved machines (on our test platform we used `Java` version 1.6 update 16 and OpenSSH 5.3). In order to install **Hadoop** we first downloaded the compressed archive of the framework following the instructions on the release website (`http://hadoop.apache.org/common/releases.html`) and then uncompressed the contents to a directory on a file system which is accessible by all machines running **Hadoop** (subsequently referred to as `HADOOP_HOME`). For example on a single workstation this directory is usually located somewhere on the corresponding local disk. In case of a cluster of workstation, this directory is usually located on a network file system (e.g., NFS). Several components of the **Hadoop** framework need to know the path to the installation directory, thus it is specified on each machine via the environment variable `HADOOP_HOME` (this variable has to be added to the user's environment on each machine if it is not done automatically).

Subsequently, few to several changes have to be made in configuration files located in the `$HADOOP_HOME/conf` directory depending on the desired operating mode. First, the path to a working `Java` environment has to be specified in '`hadoop-env.sh`'. Second, since we want to operate the framework in *pseudo distributed* (i.e., use the HDFS facilities on a single workstation) or *fully distributed* mode, one needs to set up the configuration files '`core-site.xml`', '`mapred-site.xml`', '`masters`', and '`slaves`'. An example configuration of a pseudo distributed system can be found in Appendix B.

Alternatively, on Debian GNU/Linux systems the **Hadoop** distribution can be installed from the *unstable* repository very easily using the package manager **aptitude**.

```
aptitude install hadoop-namenoded hadoop-datanoded hadoop-tasktrackerd \
hadoop-jobtrackerd hadoop-secondarynamenoded
```

This completely installs **Hadoop** and corresponding tools without further interaction needed. Nevertheless, the HDFS has to be configured as described below. Pre-configured packages for other Linux distributions can be obtained e.g., from `http://www.cloudera.com/`.

To verify the installation one can execute `hadoop version` on the command line which returns the version number of the installed software package.

A.2. Hadoop distributed file system

The final step before running **Hadoop** jobs in a pseudo distributed environment involves formatting the HDFS (the default system path on Debian is '`/var/lib/hadoop/cache`'). The command `hadoop namenode -format` serves for this purpose.

The configured **Hadoop** cluster can be started via two alternative approaches. In the first approach one can use the command `$HADOOP_HOME/bin/start-all.sh` on the command line (non-Debian) or via daemon startup scripts in '/etc/init.d' (Debian or cloudera packages). This is especially useful if an integration to cluster grid engines is desired in order to automate the startup process. In the second approach the **Hadoop** cluster can be started directly within R using the `hive_create()` and `hive_start()` functions in **hive**. The resulting '`hive`' object, representing the information about the configured cluster is stored for further use with the help of the function `hive()`. Usually, this is done automatically when the package loads, given that the **Hadoop** framework is referenced to via the `HADOOP_HOME` environment variable or if the executables are in the `PATH` and configurations are put in '`/etc/hadoop`' (as is with the Debian and cloudera packages). However, there is a known issue with IPv6 (see http://wiki.apache.org/hadoop/HadoopIPv6). Thus, if the **Hadoop** framework does not start correctly setting the configuration option `net.ipv6.bindv6only` to `0` in '`/etc/sysctl.d/bindv6only.conf`' will help.

```
R> library("hive")
R> hadoop_home <- Sys.getenv("HADOOP_HOME")
R> hive(hive_create(hadoop_home))
R> hive()

HIVE: Hadoop Cluster
- Avail. datanodes: 1
'- Max. number Map tasks per datanode: 2
'- Configured Reducer tasks: 1

R> summary(hive())

HIVE: Hadoop Cluster
- Avail. datanodes: 1
'- Max. number Map tasks per datanode: 2
'- Configured Reducer tasks: 1
---
- Hadoop version: 0.20.2
- Hadoop home/conf directory: /home/theussl/lib/hadoop/hadoop-0.20.2
- Namenode: localhost
- Datanodes:
'- localhost

R> hive_is_available()

[1] FALSE

R> hive_start()
R> hive_is_available()

[1] TRUE
```

When the **Hadoop** cluster is up and running one can retrieve **Hadoop** status and job information by visiting specific websites provided by the built-in web front end, two of them are of higher interest. Assuming that **Hadoop** is configured to run on `localhost` the websites can be accessed via a standard web browser opening `http://localhost:50030` (JobTracker) and `http://localhost:50070` (NameNode), respectively. From the former one can retrieve information about running, completed or failed jobs. We found the log files showing the error output for the distributed batch jobs very useful as they helped us debugging the R scripts generated by the **hive** package for each mapper and reducer. In the latter website one can inspect the configuration of the HDFS and browse through the file system.

The HDFS can be accessed directly in R with `DFS_*()` functions.

```
R> DFS_list("/")

[1] "home" "tmp"

R> DFS_dir_create("/tmp/test")
R> DFS_write_lines(c("Hello HDFS", "Bye Bye HDFS"), "/tmp/test/hdfs.txt")
R> DFS_list("/tmp/test")

[1] "hdfs.txt"

R> DFS_read_lines("/tmp/test/hdfs.txt")

[1] "Hello HDFS"   "Bye Bye HDFS"
```

For a complete reference to implemented `DFS_*()` functions see the **hive** help pages.

B. Hadoop configuration

We recommend to set the following parameters in the corresponding configuration files in order to set up pseudo distributed mode on a single machine with two processors/cores. In case of a Debian installation the files '`masters`' and '`slaves`' have to be created in the '`/etc/hadoop/conf`' directory (each containing `localhost`).

'`core-site.xml`' contains:

```
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:9000</value>
  </property>
  <property>
    <name>hadoop.tmp.dir</name>
    <value>/home/${user.name}/tmp/hadoop-${user.name}</value>
  </property>
</configuration>
```

'`mapred-site.xml`' contains:

```
<configuration>
<property>
    <name>mapred.job.tracker</name>
    <value>hdfs://localhost:9001</value>
  </property>
  <property>
    <name>mapred.tasktracker.map.tasks.maximum</name>
    <value>2</value>
    <description>The maximum number of map tasks that will be run
    simultaneously by a task tracker.
    </description>
  </property>
  <property>
    <name>mapred.tasktracker.reduce.tasks.maximum</name>
    <value>2</value>
    <description>The maximum number of reduce tasks that will be run
    simultaneously by a task tracker.
    </description>
  </property>
</configuration>
```

'`hdfs-site.xml`' contains:

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
</configuration>
```

'`masters`' contains the IP address of the master servers (or `localhost`) to this file.

'`slaves`' contains the IP address of all data nodes (or `localhost`) to this file.

Affiliation:

Kurt Hornik, Stefan Theußl
Institute for Statistics and Mathematics
WU Wirtschaftsuniversität Wien
Augasse 2–6
1090 Wien, Austria
E-mail: `Kurt.Hornik@R-project.org`, `Stefan.Theussl@R-project.org`

Ingo Feinerer
Institute of Information Systems
Technische Universität Wien
1040 Wien, Austria
E-mail: `Ingo.Feinerer@tuwien.ac.at`

Journal of Statistical Software `http://www.jstatsoft.org/`
published by the American Statistical Association `http://www.amstat.org/`

Volume 51, Issue 5 *Submitted:* 2011-03-16
November 2012 *Accepted:* 2012-11-02