Importing and exporting data

Claudius Gräbner-Radkowitsch

2022-03-09

Contents


```
knitr::opts_chunk$set(echo = TRUE, eval = FALSE)
```

0.1 Daten einlesen und schreiben

0.1.1 Einlesen von Datensätzen

Wenige Arbeitsschritte können so frustrierend sein wie das Einlesen von Daten. Sie können sich gar nicht vorstellen was hier alles schiefgehen kann! Aber kein Grund zur übertriebenen Sorge: wir können viel Frustration vermeiden wenn wir am Anfang unserer Karriere ausreichend Zeit in die absoluten Grundlagen von Einlesefunktionen investieren. Also, auch wenn die nächsten Zeilen etwas trocken wirken: sie werden Ihnen später viel Zeit ersparen!

Das am weitesten verbreitete Dateiformat ist csv. 'csv' steht für 'comma separated values' und diese Dateien sind einfache Textdateien, in denen Spalten mit bestimmten Symbolen, in der Regel einem Komma, getrennt sind. Aufgrund dieser Einfachheit sind diese Dateien auf allen Plattformen und quasi von allen Programmen ohne Probleme lesbar.

In R gibt es verschiedene Möglichkeiten csv-Dateien einzulesen. Die mit Abstand beste Option ist dabei die Funktion `data.table::fread()` aus dem Paket `data.table`, da sie nicht nur sehr flexibel spezifiziert werden kann, sondern auch deutlich schneller als andere Funktionen arbeitet.

Wir gehen im Folgenden davon aus, dass wir die Datei `data/tidy/export_daten.csv` einlesen wollen. Die Datei sieht folgendermaSSen aus:

```
iso2c,year,Exporte
AT,2012,53.97
AT,2013,53.44
AT,2014,53.38
```

Es handelt sich also um eine sehr standardmäSSige csv-Datei, die wir einfach mit der Funktion `data.table::fread()` einlesen können. Dazu übergeben wir `data.table::fread()` nur das einzige wirklich notwendige Argument: den Dateipfad. Der besseren Übersicht halber sollte dieser immer separat definiert werden:

```
daten_pfad <- here::here("data/tidy/export_daten.csv")
daten <- data.table::fread(daten_pfad)
daten
```

Vielleicht fragen Sie sich wie `data.table::fread()` die Spalten bezüglich ihres Datentyps interpretiert hat? Das können wir folgendermaSSen überprüfen:

```
typeof(daten[["year"]])
```

In der Regel funktioniert die automatische Typerkennung von `data.table::fread()` sehr gut. Ich empfehle dennoch die Typen im Zweifel manuell zu spezifizieren, aus folgenden Gründen: (1) Sie merken leichter wenn es mit einer Spalte ein Problem gibt, z.B. wenn in einer Spalte, die ausschlieSSlich aus Zahlen besteht ein Wort vorkommt. Wenn Sie diese Spalte nicht manuell als `double` spezifizieren würden, würde `data.table::fread()` sie einfach still und heimlich als `character` interpretieren und Sie wundern sich später, warum Sie für die Spalte keinen Durchschnitt berechnen können; (2) Ihr Code wird leichter lesbar; und (3) der Einlesevorgang wird deutlich beschleunigt da `data.table::fread()` die Typen nicht selbst 'erraten' muss.

Sie können die Spaltentypen manuell über das Argument `colClasses` einstellen, indem Sie einfach einen Vektor mit den Datentypen angeben:

```
daten_pfad <- here::here("data/tidy/export_daten.csv")
daten <- data.table::fread(
  daten_pfad, colClasses = c("character", "double", "double")
  )
typeof(daten[["year"]])
```

Da es bei sehr groSSen Dateien einen extremen Unterschied macht ob Sie die Spaltentypen angeben oder nicht macht es in einem solchen Fall häufig Sinn, zunächst mal nur die erste Zeile des Datensatzes einzulesen, sich anzuschauen welche Typen die Spalten haben sollten und dann den gesamten Datensatz mit den richtig spezifizierten Spaltentypen einzuladen. Sie können nur die erste Zeile einladen indem Sie das Argument `nrows` verwenden:

```
daten_pfad <- here("data/tidy/export_daten.csv")
daten <- data.table::fread(
  daten_pfad, colClasses = c("character", "double", "double"),
  nrows = 1)
daten
```

Manchmal möchten Sie auch nur eine bestimmte Auswahl an Spalten einlesen. Auch das kann bei groSSen Datensätzen viel Zeit sparen. Wenn wir nur das Land und die Anzahl der Exporte haben wollen, spezifizieren wir das über das Argument `select`:

```
daten_pfad <- here::here("data/tidy/export_daten.csv")
daten <- data.table::fread(
  daten_pfad, colClasses = c("character", "double", "double"),
  nrows = 1,
  select = c("iso2c", "Exporte")
  )
daten
```

Die Beispiel-Datei oben war sehr angenehm formatiert. Häufig werden aber andere Spalten- und Dezimalkennzeichen verwendet. Gerade in Deutschland ist es verbreitet, Spalten mit ; zu trennen und das Komma als Dezimaltrenner zu verwenden. Unsere Beispiel-Datei oben sähe dann so aus:

```
iso2c;year;Exporte
AT;2012;53,97
AT;2013;53,44
AT;2014;53,38
```

Zum Glück können wir das Spaltentrennzeichen über das Argument `sep` und das Kommatrennzeichen über das Argument `dec` manuell spezifizieren:[1]

```
daten_pfad <- here::here("data/tidy/export_daten_dt.csv")
daten <- data.table::fread(
  daten_pfad, colClasses = c("character", "double", "double"),
  sep = ";",
  dec = ","
  )
daten
```

`data.table::fread()` verfügt noch über viele weitere Spezifizierungsmöglichkeiten, mit denen Sie sich am besten im konkreten Anwendungsfall vertraut machen. Auch ein Blick in die Hilfeseite ist recht illustrativ. Für die meisten Anwendungsfälle sind Sie jetzt aber gut aufgestellt.

> **Anmerkungen zu komprimierten Dateien:** Häufig werden Sie auch komprimierte Dateien einlesen wollen. Gerade komprimierte csv-Dateien kommen häufig vor. In den meisten Fällen können Sie diese Dateien direkt mit `data.table::fread()` einlesen. Falls nicht, können Sie `data.table::fread()` aber auch dem entsprechenden UNIX-Befehl zum Entpacken als Argument `cmd` übergeben, also z.B. `data.table::fread("unzip -p data/gezipte_daten.csv.bz2")`. Weitere Informationen finden Sie sehr einfach im Internet.

[1] Auch hier gilt, dass die automatische Erkennung von `data.table::fread()` schon sehr gut funktioniert, aber die manuelle Eingabe immer sicherer und transparenter ist.

Auch wenn csv-Dateien die am weitesten verbreiteten Daten sind: es gibt natürlich noch viele weitere Formate mit denen Sie in Kontakt kommen werden. Hier möchte ich exemplarisch auf drei weitere Formate (`.rds`, `.rdata` und `.dta`) eingehen:

R verfügt über zwei 'hauseigene' Formate, die sich extrem gut zum Speichern von gröSSeren Daten eignen, aber eben nur von R geöffnet werden können. Diese Dateien enden mit `.rds`, bzw. mit `.RData` oder `.Rda`, wobei `.Rda` nur eine Abkürzung für `.RData` ist.

Dabei gilt, dass `.rds`-Dateien einzelne R-Objekte enthalten, z.B. einen einzelnen Datensatz, aber auch jedes andere Objekt (Vektor, Liste, etc.) kann als `.rds`-Dateie gespeichert werden. Solche Dateien können mit der Funktion `readRDS()` gelesen werden, die als einziges Argument den Dateinamen annimmt:

```
daten_pfad <- here::here("data/tidy/export_daten.rds")
daten <- readRDS(daten_pfad)
daten
```

`.RData`-Dateien können auch mehrere Objekte enthalten. Zudem gibt die entsprechende Funktion `load()` kein Objekt aus, dem Sie einen Namen zuweisen können. Vielmehr behalten die Objekte den Namen, mit dem sie ursprünglich gespeichert wurden. In diesem Fall wurden in der Datei `data/tidy/test_daten.RData` der Datensatz `test_dat` und der Vektor `test_vec` gespeichert. Entsprechend sind sie nach dem Einlesen verfügbar:

```
load(here::here("data/tidy/test_daten.RData"))
test_dat
test_vec
```

Die Verwendung von `.RData` ist besonders dann hilfreich, wenn Sie mehrere Objekte speichern wollen und wenn einige dieser Objekte keine Datensätze sind, für die auch andere Formate zur Verfügung stehen.

Ein in der Ökonomik häufig verwendetes Format ist das von der Software STATA verwendete Format `.dta`. Um Dateien in diesem Format lesen zu können verwenden Sie die Funktion `read_dta()` aus dem Paket haven [@R-haven], die als einziges Argumente den Dateinamen akzeptiert:

```
dta_datei <- here::here("data/tidy/export_daten.dta")
dta_daten <- haven::read_dta(dta_datei)
head(dta_daten, 2)
```

Das Paket haven stellt auch Funktionen zum Lesen von SAS oder SPSS-Dateien bereit.

> **Exkurs: GroSSe Zahlen und Probleme mit `int64`** Wir haben in Kapital @ref(basics) ja bereits den etwas besonderen Datentyp `bit64::integer64` kennen gelernt. Da dieser Datentyp mit einigen Operatoren inkompatibel ist und merkwürdiges Verhalten verursacht wenn das Paket `bit64` nicht installiert ist, sollten wir seine Verwendung unbedingt vermeiden. Wenn Sie aber mit `data.table::fread` einen Datensatz einlesen, der ganze Zahlen

beinhaltet, die gröSSer sind als 2147483647, dann werden diese automatisch als `bit64::integer64` kodiert.[2]

```
large_nb_frame <- data.table::fread(
  here::here("data/tidy/BIPKonsum.csv"))
str(large_nb_frame, vec.len=3)
```

Das können wir verhindern, indem wir die Spaltentypen explizit über `colClasses` spezifizieren oder aber zur Sicherheit das Argument `integer64` auf `"double"` setzen:

```
large_nb_frame <- data.table::fread(
  here::here("data/tidy/BIPKonsum.csv"),
  integer64 = "double")
str(large_nb_frame, vec.len=3)
```

0.1.2 Speichern von Daten

Im Vergleich zum Einlesen von Daten ist das Speichern deutlich einfacher, weil sich die Daten ja bereits in einem vernünftigen Format befinden. Die gröSSte Frage hier ist also: in welchem Dateiformat sollten Sie Ihre Daten speichern?

In der groSSen Mehrheit der Fälle ist diese Frage klar mit `.csv` zu beantworten. Dieses Format ist einfach zu lesen und absolut plattformkompatibel. Es hat auch nicht die schlechtesten Eigenschaften was Lese- und Schreibgeschwindigkeit angeht, insbesondere wenn man die Daten komprimiert.

Die schnellste und meines Erachtens mit Abstand beste Funktion zum Schreiben von csv-Dateien ist die Funktion `fwrite()` aus dem Paket `data.table`. Angenommen wir haben einen Datensatz `test_data`, den wir im Unterordner `data/tidy` als `test_data.csv` speichern wollen. Das geht mit `data.table::fwrite()` ganz einfach:

```
datei_name <- here::here("data/tidy/test_data.csv")
data.table::fwrite(test_data, file = datei_name)
```

Neben dem zu schreibenden Objekt als erstem Argument benötigen Sie noch das Argument `file`, welches den Namen und Pfad der zu schreibenden Datei spezifiziert. Der Übersicht halber ist es oft empfehlenswert diesen Pfad zuerst als `character`-Objekt zu speichern und dann an die Funktion `data.table::fwrite()` zu übergeben.

`data.table::fwrite()` akzeptiert noch einige weitere optionale Argumente, die Sie im GroSSteil der Fälle aber nicht benötigen. Schauen Sie bei Interesse einfach einmal in die Hilfefunktion!

Falls Ihr Datensatz im csv-Format doch zu groSS ist, Sie aber aufgrund von Kompatibilitätsanforderungen kein spezialisiertes Format benutzen wollen, bietet es sich an die csv-Datei zu komprimieren. Natürlich könnten Sie das händisch in Ihrem Datei-Explorer

[2] Wenn Sie das Paket `bit64` nicht installiert haben, bekommen Sie zudem eine etwas merkwürdig anmutende Warnung zu lesen.

machen, aber das ist vollkommen überholt. Sie können das gleich in R miterledigen indem Sie z.B. die Funktion `gzip()` aus dem Paket `R.utils` [@R-R.utils] verwenden:

```
csv_datei_name <- here::here("data/tidy/test_data.csv")
data.table::fwrite(test_data, file = csv_datei_name)
R.utils::gzip(
  csv_datei_name,
  destname=paste0(csv_datei_name, ".gz"),
  overwrite = TRUE, remove=TRUE)
```

Diese Funktion akzeptiert als erstes Argument den Pfad zu der zu komprimierenden Datei, also zweites Argument (`destname`) den Namen, den die komprimierte Datei tragen soll und einige weitere optionale Argumente. Häufig bietet sich `overwrite = TRUE` an, um alte Versionen der komprimierten Datei im Zweifel zu überschreiben, und `remove=TRUE` um die un-komprimierte Datei nach erfolgter Komprimierung zu löschen.

> **Hinweise zu verschiedenen zip-Formaten:** Die Funktion `R.utils::gzip()` komprimiert eine Datei mit dem GNU zip Algorithmus. Die resultierende komprimierten Dateien sollten mit der zusätzlichen Endung `.gz` gekennzeichnet werden. `R.utils::gzip()` ist eine relativ schnell arbeitende Funktion, allerdings mit mäSSigen Kompressionseigenschaften. Wenn Sie bereit sind längere Arbeitszeit für ein besseres Kompressionsergebnis in Kauf zu nehmen, sollten Sie sich die Funktion `R.utils::bzip2()` ansehen, welche den bzip2-Algorithmus implementiert. Dieser hat eine deutlich bessere Kompressionsrate (die komprimierten Dateien sind also deutlich kleiner), allerdings ist `R.utils::bzip2()` auch deutlich langsamer als `R.utils::gzip()`. Dateien, die mit `R.utils::bzip2()` komprimiert wurden, sollten mit der Endung `.bz2` gekennzeichnet werden. Entsprechend sieht der Code von oben mit `R.utils::bzip2()` anstatt `R.utils::gzip()` folgendermaSSen aus:

```
csv_datei_name <- here::here("data/tidy/test_data.csv")
data.table::fwrite(test_data, csv_datei_name)
R.utils::bzip2(
  csv_datei_name,
  destname=paste0(csv_datei_name, ".bz2"),
  overwrite = TRUE)
```

> Einen Vergleich der Kompressionseigenschaften und Lese- und Schreibgeschwindigkeiten ist immer auch kontextabhängig, im Internet finden sich viele Diskussionen zu dem Thema. Am Anfang sind Sie mit `R.utils::gzip()` und `R.utils::bzip2()` aber eigentlich für alle relevanten Fälle gut aufgestellt.

Die oben bereits vorgestellten R-spezifischen Formate `.Rdata` und `.rds` verfügen über deutliche Geschwindigkeits- und Komprimierungsvorteile gegenüber dem csv-Format und sind dabei trotzdem vollkommen plattformkompatibel. Einziger Nachteil: alle Irren, die nicht R benutzen, können Ihre Daten nicht öffnen. Manchmal mag das eine verdiente Strafe, manchmal aber auch ein Ausschlusskriterium sein.

```
saveRDS(object = test_data,  file = here::here("data/tidy/export_daten.rds"))
```

Wie Sie sehen sind zwei Argumente zentral: das erste Argument, `object` spezifiziert das zu speichernde Objekt und `file` den Dateipfad. Darüber hinaus können Sie mit dem optionalen Argument `compress` hier die Kompressionsart auswählen. Ähnlich wie oben gilt, dass `gz` am schnellsten und `bz` am stärksten ist. `xz` liegt in der Mitte.

Wenn Sie mehrere Objekte auf einmal speichern möchten können Sie das über das Format `.RData` machen. Die entsprechende Funktion ist `save()`. Zwar können Sie einfach alle zu speichernden Objekte als die ersten Argumente an die Funktion übergeben, es ist aber übersichtlicher das über das Argument `list` zu erledigen. Der folgende Code speichert die beiden Objekte `test_data` und `daten` in der Datei `"data/tidy/datensammlung.Rdata"`:

```
save(
  list=c("test_data", "daten"),
  file=here::here("data/tidy/datensammlung.RData")
  )
```

Wie `saveRDS()` können Sie bei `save()` über das Argument `compress` den Kompressionsalgorithmus auswählen, allerdings können Sie mit `compression_level` zusätzlich noch die Stärke von `1` (schnell, aber wenig Kompression) bis `9` (langsamer, aber starke Kompression) auswählen.

Da, wie oben erwähnt, gerade in der Ökonomik auch häufig mit der kostenpflichtigen Software STATA gearbeitet wird, möchte ich noch kurz erkläutern, wie man einen Datensatz im STATA-Format `.dta` speichern kann. Dazu verwenden wir die Funktion `write_dta()` aus dem Paket haven.

```
haven::write_dta(
  test_data, here::here("data/tidy/test_daten.dta"))
```

Für `SAS`- und `SPSS`-Daten gibt es ähnliche Funktionen, die ebenfalls durch das haven-Paket bereitgestellt werden.

> **Hinweis:** Gerade bei groSSen Datensätzen kommt es wirklich sehr auf die Lese- und Schreibgeschwindigkeit von Funktionen an. Auch stellt sich hier die Frage nach dem besten Dateiformat noch einmal viel deutlicher als das bei kleinen Datensätzen der Fall ist und sich die Formatfrage vor allem um das Thema 'Kompatibilität' dreht. Einige nette Beiträge, die verschiedene Funktionen und Formate bezüglich ihrer Geschwindigkeit vergleichen finden Sie z.B. hier oder
> hier.