AZ R PROGRAMOZÁS ALAPJAI

ILLÉS FERENC KERESZTÚRI JUDIT LILLA

Budapesti Corvinus Egyetem

2018

ISBN: 978-963-503-744-5

Előszó

Ez a könyv Keresztúri Judit Lilla – Antal Beáta – Illés Ferenc: Bevezetés az R programozásba című, a Vállalati pénzügyi információs rendszerek tárgyhoz két évvel ezelőtt írott jegyzet végleges verziójának tekinthető.

A Vállalati Pénzügyi Információs Rendszerek tárgy a Budapesti Corvinus Egyetem Pénzügy mesterszak Vállalati pénzügyek specializáció utolsó előtti vagy utolsó félévének kötelező tantárgya, melynek célja megismertetni a hallgatókat a vállalatok pénzügyi funkcióival, a pénzügyi vezetés feladataival, valamint a vállalati pénzügyi döntéseket támogató információs rendszerekkel.

A tanagyag keretében évek óta nagy súlyt kap az R programozási nyelv oktatása, melynek alapszintű ismerete elengedhetetlen feltétele a tárgy teljesítésének. Mivel a tárgyi tematika jelentős átalakuláson esett át az elmúlt években, a végső fázisba érve célszerűnek tűnik a korábban kiadott jegyzetet kiegészítve és tartalmát a tárgy tematikájára fókuszálva ismét kiadni, hogy egy bővebb és frissebb segédanyag álljon a hallgatók rendelkezésére.

A könyv kifejezetten pénzügy mesterszakos hallgatók számára készült, a szakon oktatott tárgyak során elsajátított előismeretekre épül, ezért ne tekintsük önállóan feldolgozható programozási könyvnek, sem az órákon való aktív részvétel helyettesítőjének.

A mintapéldákhoz használt fájlok a `http://www.uni-corvinus.hu/~lkeresz` linken érhetőek el.

A jelenlegi kiadást tartalmában többé-kevésbé véglegesnek szánjuk, azonban legjobb szándékaink ellenére is bizonyára maradtak benne hibák, hiányosságok, pontatlanságok, amelyekre vonatkozó észrevételeket a `lilla.kereszturi@uni-corvinus.hu` e-mail címen köszönettel fogadunk.

Noha önálló feldolgozásra nem ajánljuk, a könyv szabadon terjeszthető, módosítható a szöveg integritásának megőrzésével (azaz az esetleges további önálló szövegrészek, módosítások és az eredeti forrás jól elkülöníthető megjelölésével), és akár részleteiben, akár teljes terjedelmében felhasználható oktatási célra és egyéb non-profit tevékenységre. A szabad felhasználás nem terjed ki jövedelemszerző- és/vagy jövedelemfokozó tevékenységre.

2018. október - november

A szerzők

Tartalomjegyzék

1. fejezet

Mi az R?

„R is a language and environment for statistical computing and graphics”[1]. Tehát az R egyrészt egy programozási nyelv, másrészt egy futtatókörnyezet, azaz olyan szoftver, mely az R nyelven írt kódot értelmezni és futtatni képes. Az R ingyenes és open source. A nyelv és a szoftver fejlesztését egy non-profit szervezet a The R Foundation for Statistical Computing végzi. Erről bővebb infó a https://www.r-project.org/about.html és https://www.r-project.org/foundation/ oldalon található.

1.1. Telepítés

Az R project kezdőoldala: https://www.r-project.org/. A szoftver letöltéséhez kattintsunk a download-ra, majd válasszuk ki Magyarországot, ekkor ide jutunk: http://cran.rapporter.net/. A szoftver telepítése a download ⇒ next ⇒ next ⇒ next ⇒ ... ⇒ finish módszerrel könnyen elvégezhető, ha elfogadjuk az alapértelmezett beállításokat.

Ha elindítjuk, láthatjuk, hogy a szoftver elég fapados, egy egyszerű command-line interface-t, kapunk, ez azt jelenti, hogy nincs grafikus felület (GUI), mint pl. az SPSS-ben, csak egy konzol, ahova beírhatjuk a futtatandó utasításokat. Próbáljuk ki! Ha beírjuk, hogy `3+2` és megnyomjuk az entert, kiszámolja és visszaírja az eredményt. Ezen örvendezve zárjuk is be a programot.

Mivel az egyszerű R konzolban programozni nem nagy felhasználói élmény, több olyan kiegészítő program is létezik, mely a használatát valamelyest megkönnyíti. Ezek közül egyedül az RStudio-t fogjuk használni.

Az RStudio szintén ingyenes, letölthető a https://www.rstudio.com/ weboldalról. Fontos tudni, hogy az RStudio csak egy code editor, más szóval egy szövegszerkesztő program, ami az R futtatókörnyezet nélkül, önállóan nem működik. Legfőbb funkciói: kiemeli az összetartozó zárójeleket, felismeri és kiszínezi a kulcsszavakat, az első pár karakter alapján kitalálja és kiegészíti az R függvények nevét, stb., tehát elsősorban kényelmi funkciója van, kódot futtatni nem képes. Hatékonyan tudja azon-

[1] `https://www.r-project.org/about.html`

ban gátolni a program alapból hibátlan működését, mert előszeretettel fagy le számításigényes algoritmus futtatásakor. Ez a hiba nem az R-ben, hanem az RStudioban van, az okát pontosan nem ismerjük, az újabb verziók bizonyára kiküszöbölik majd.

1.2. RStudio használata

Az RStudio indulásakor egy három vagy négy ablakból álló képernyőt látunk. Alapértelmezett beállítás esetén a baloldali vagy bal alsó ugyanazt a funkciót látja el, mint az R konzol, ide lehet beírni az azonnal futtatandó utasításokat. A másik három ablak az RStudio „extra szolgáltatása". A jobb felső a Global Environment, itt láthatjuk az általunk létrehozott változókat, objektumokat, függvényeket, beolvasott, vagy generált adatokat. Az adatot tartalmazó objektumokra rákattintva azt táblázatos formában meg is jeleníti a konzol fölötti területen. A jobboldali alsó ablakban látjuk többek között a help-et és a létrehozott ábrákat. A negyedik ablak maga a szövegszerkesztő, mely csak akkor látszik, ha meg van benne nyitva legalább egy fájl. Az R kódokat általában „.R" kiterjesztésű egyszerű szövegfájlokba írjuk, melyeket létrehozhatunk akár sima notepad-ben is, vagy az RStudio-ban is. Ehhez nyomjuk meg a ctrl+shift+N billentyűkombinációt, vagy használjuk a File ⇒ New File ⇒ R script menüt. Ha már van egy ilyen fájlunk, akkor azt a ctrl+O billentyűkombinációval vagy a File ⇒ Open File menüponttal nyithatjuk meg. Ekkor megnyílik az editor a bal felső ablakban, és láthatjuk, futtathatjuk, illetve szerkeszthetjük a benne lévő kódot. Az RStudio fenti layout-ja a Tools ⇒ Global Options –ben a Panel Layout fülön módosítható.

Az R kód általában adatbeolvasásból és függvények definícióiból áll. Ezeket célszerű összetartozó logikai egységenként egy-egy .R fájlban tárolni, a megfelelőt mindig betölteni, és a konzolba csupán a legegyszerűbb utasításokat (például függvényhívásokat) beírni. Az RStudio nagyon megkönnyíti ezek használatát, mert az editorban egyszerre több kódfájl is megnyitható, szerkeszthető, és a benne lévő utasítások egyben vagy blokkonként lefuttathatók. Az editor ablakának felső szegélyén lévő menüben a jobb felső sarokban található egymás mellett a Run (zöld nyíl), a re-Run és a Source (kék nyíl) utasítás. A Source gombra kattintás (ami ekvivalens a source függvény meghívásával, mely a konzolban ilyenkor meg is jelenik) a fájlban lévő összes utasítást végrehajtja, beleértve a változók és függvények definiálását (ez nem jelenti a függvények meghívását, ha a fájlban nincs explicit függvényhívás, de a függvények és minden létrehozott objektum megjelenik a global environment-ben, és onnantól kezdve létezik és elérhető). A Run gombra kattintás az RStudio legnagyobb találmánya, mert ez lefuttatja azt a sort, amelyben a kurzor áll, vagy a kijelölt kódrészletet, ha valami ki van jelölve. Az RStudio folyamatosan menti (és megjegyzi) a konzolba beírt utasításokat, melyek a beírás fordított sorrendjében (vagyis a legutolsóval kezdve) visszaírhatók a konzolba a felfelé (és lefelé) nyíl nyomogatásával, így tömérdek felesleges gépelést lehet megspórolni.

Amint elindítjuk az R programot (akár az RStudioból, akár anélkül) a nyitóképernyőn mindig ugyanazt a szöveget olvashatjuk, mely tudatja a használt verzió számát, illetve, hogy az R free és

mindenki csak saját felelősségére használhatja, valamint, hogy hogyan lehet megnyitni a help-et. Az R dokumentáltsága kifejezetten magas színvonalú. Ha beírunk a konzolba egy kérdőjelet és utána egy elérhető függvény nevét, azonnal megnyitja az RStudio jobb alsó ablakában, vagy egy böngészőben a rá vonatkozó dokumentációt, mely tartalmazza a függvény leírását, a hivatkozott objektumok linkjeit, egy ”See Also” ajánlott hivatkozásokból álló részt, és példákat. Ha rosszul írjuk be a keresett kifejezést, nem fog találni semmit, de szinte biztos, hogy bármi legyen is a problémánk, R-es fórumokon azt valaki már megkérdezte és valaki megválaszolta, érdemes tehát Google-ben is keresgélni, ha elakadunk valamivel. A `help.start()` függvény meghívásával megnyithatjuk az R hivatalos dokumentációját és a User Manual-t. Egy bevezető jegyzetet találhatunk többek között a https://cran.r-project.org/doc/manuals/R-intro.pdf oldalon.

2. fejezet

Package-ek

Az R-ben elképesztő mennyiségű lineáris algebrai, statisztikai, adatbányászati modell, módszer, algoritmus van, az egyszerű lineáris regressziótól a neurális hálókon át a szövegbányászati eljárásokig, sőt számtalan minta-adatbázis és még az ég tudja mi minden. Nyilvánvaló, hogy nincs minden egyes felhasználónak szüksége az összes lehetséges (valószínűleg több ezer gigabájtot kitevő) funkcióra. Ezért a felhasználás szempontjából összetartozó eljárások és (néha) adatok egymástól (többé-kevésbé) függetlenül telepíthető és felhasználható csomagokba úgynevezett ”package”-ekbe kerülnek. A package-ek nagy részét ma már maguk a felhasználók írják és bocsátják a közösség (azaz egymás) rendelkezésére (ennek minden előnyével és hátrányával együtt). A package-ek, az alapcsomaggal (base package) és az egymással való kompatibilitás megőrzése érdekében meghatározott struktúrában és előírt minimális dokumentációval ellátva kerülnek fel a CRAN (The Comprehensive R Archive Network) weboldalra. Ennek köszönhetően felhasználásuk rendkívül egyszerű, mert az R automatizáltan le tudja tölteni és telepíteni a repository-ból bármelyik package-t. A cran.rapporter.net oldalon fent van az összes package a bal oldali menüben név és dátum szerint is rendezve. Kattintsunk itt rá pl. a matrixStats package-re. Minden R-es package-nek egy ehhez hasonló kinézetű weboldala van. Innen le lehet tölteni a .zip fájlt, és a reference manualt, meg lehet nézni a dependency-ket, vagyis, hogy az adott package mely más package-eket használ fel, melyek nélkül nem működik. Fel lehet telepíteni a package-t a .zip fájlból közvetlenül, ha letöltöttük, de ritkán szokták a package-eket egyesével manuálisan telepíteni. Az install.packages utasítás feltelepíti, azaz letölti és kicsomagolja a megfelelő könyvtárba, ahol később megtalálja, a package-eket a dependency-kel együtt. A `library` (vagy `require`) utasítás betölti a package-t, azaz elérhetővé teszi a benne lévő függvényeket és (esetleges) adatokat. A package installálása és betöltése nem ugyanaz, installálni egy gépre csak egyszer kell, betölteni viszont minden használat előtt szükséges. A már korábban feltelepített package betöltéséhez nincs szükség internetkapcsolatra. Az egyes package-ek árnyékolhatják egymást, amiről az R kiír egy warning-ot. Telepítsük föl és töltsük be például a `gmp` package-t, az üzenet szerint az apply függvény ”masked from package base”. Ez szerencsére nem jelenti azt, hogy az összes függvényt, ami ezeket használja, elrontottunk, csak azt, hogy van két azonos nevű függvény, melyek működési elvére később visszatérünk.

3. fejezet

Változók, adattípusok és elemi adatstruktúrák

Hozzunk létre egy változót! Írjuk be a konzolba, hogy

```
a <- 1
```

Az utasítás jelentése: „az `'a'` változó értéke legyen egy 1 hosszúságú vektor, amelynek első (és egyetlen) koordinátája az 1 valós szám”. R-ben a változóknak való értékadás nem az = jellel, hanem a

```
változó neve <- kifejezés
```

utasítással történik, aminek előnye, hogy nem szimmetrikus, így megfordítható, tehát az `1 -> a` utasítás ugyanezt jelenti. A létrehozott változó neve és értéke megjelenik a Global Environment-ben a jobb felső ablakban. Ki is listázhatjuk az összes változót az `ls` függvénnyel. Amelyik változóra már nincs szükségünk, azt eltávolíthatjuk a memóriából (és nagyobb adathalmaz esetén érdemes is, mert az R magától nem töröl változókat a workspace-ből) az `rm` függvénnyel. (A függvények lokális változói nem listázhatóak a Global Environment-ben, és maguktól megsemmisülnek, ha a függvény lefutott.)

Az R-ben nem kell és nem is lehet a változókat deklarálni a típusuk megadásával. Ennek ellenére minden változónak van típusa („általános” típus, mint pl. a VBA-ban a „Variant” nem létezik), ami a változóval együtt az értékadáskor jön létre. Az R alaptípusai a `logical, integer, numeric, character,` és a `complex`.

Írjuk be a konzolba, hogy `class(a)`, ezzel lekérdezhetjük a változó típusát. Ha beírjuk, hogy `is.integer(a)`, kiírja, hogy `FALSE`, ami elég bosszantó, mert az 1 egész szám. Most írjuk be, hogy `b <- 1:10`, ezzel létrehozunk egy 10 hosszúságú vektort, melynek elemei 1-től 10-ig terjednek. Ha beírjuk, hogy `is.numeric(b)`, illetve `is.integer(b)`, mindkét kérdésre `TRUE` a válasz, tehát egy változónak több típusa is lehet. Ennek ellenére az `is.complex(b)` értéke `FALSE`, pedig minden egész szám komplex szám is.

Az R meglehetősen kaotikus és kiszámíthatatlan módon kezeli a típusokat, és szükség esetén konvertálja a változókat egyik típusból a másikba (akár adatvesztés árán is!). Másrészről a változók

típusának nagy jelentősége van a velük végezhető műveletek szempontjából, mert például `sqrt(-1)` értéket nem számolja ki, de `sqrt(as.complex(-1))` értékeként megkapjuk a `0+1i`-t (ami szintén nem tökéletes, mert `0-1i` is lehetne).

Az aritmetikai és konverziós műveletek elvégezhetőségének biztosítása érdekében az R-ben speciális konstansok vannak, ezeket az `Inf, -Inf, NaN` és `NA` szimbólumok jelölik. `Inf` és `{Inf` keletkezik például pozitív, illetve negatív (de nem nulla) szám nullával való osztásának eredményeként, illetve `{Inf` keletkezik például 0 logaritmusaként. Hüvelykujjszabályként mondhatjuk, hogy végtelen és mínusz végtelen, a matematikailag „értelmes" határértékek jelölésére szolgál (azon megszorítással, hogy a 0-t pozitívnak tekintjük például az osztásnál), de emellett például $e^{1000} = \text{Inf}$ a túlcsordulás miatt nem ábrázolható, túl nagy számok esetén sem. Az `NaN` (not a number) típusa `numeric` és az értelmetlen matematikai műveletek eredményeként adódik. Értelmetlen matematikai művelet például a negatív (nem komplex típusú) számból való gyökvonás, vagy a negatív (nem komplex típusú) szám logaritmusa, illetve a 0/0 osztás. Az `NA` logikai konstans, jelentése `Not Available`, és adatsorokban a hiányzó értékek jelölésére használjuk, valamint az elvégezhetetlen típuskonverzió eredményeként is ez jön létre. Ilyen például `as.numeric("alma")`. Ezeknek köszönhetően az R – kis túlzással – minden létező adattal, minden létező műveletet el tud végezni, nem dob hibát és nem áll le a futó kód minden alkalommal, ha egy adatsorban százezer megfigyelésből három hiányzik, vagy adathibát tartalmaz vagy negatív számból gyököt kéne vonni.

3.1. Adatok és attribútumok

Az R speciális programozási nyelv, amelyet elsősorban (bár nem kizárólag) statisztikai modellek fejlesztésére és adatelemzésre, vizualizációra fejlesztettek ki. Adattárolási modelljét és eljárásait úgy találták ki, hogy elősegítsék nagyobb mennyiségű adat viszonylag gyors és egyszerű feldolgozását. A változók ritkán tárolnak egyetlen értéket, általában egy egész halmazt, mátrixot, vektort, idősort, vagy adattáblát tartalmazó változókat hozunk létre. Ez annyira igaz, hogy az R-ben nem is létezik skalár típusú változó, ami annak tűnik, az valójában 1 hosszúságú vektor.

Az R-ben létrehozott változók és adattárolás működésének alaplogikája a következő:

Változó = Adat + MetaAdat

Metaadat azt jelenti, "adat az adatról". Egy változó tehát általában a változót alkotó adatok és az adatok értelmezésének összessége. Az adatok általában összefüggő memóriaterületen helyezkednek el, hogy könnyen hozzáférhetőek legyenek, az adat értelmezését leíró "metaadat" pedig a változó úgynevezett attribútumaiban található. A legtöbb programozási nyelvben a változó értelmezéséhez szükséges összes tárolt információ a változó típusa. Az R ennél jóval szofisztikáltabb, minden változónak tetszőleges számú attribútuma lehet, mely lényegében kulcs-érték párok halmaza (ahol a kulcs egyedi) és egy-egy

fontos információt tárol a változóról. Egy mátrix és egy vektor például adattárolás szempontjából teljesen egyforma, a kettőt pusztán az különbözteti meg, hogy a mátrixnak van egy `dim` nevű attribútuma, mely egy 2-hosszúságú vektor, és a mátrix sorainak és oszlopainak számát tartalmazza.

Az R legelemibb adatstruktúrái a vektor, mátrix, és a tömb. Most ezek alaptulajdonságait tekintjük át.

3.2. Elemi adatstruktúrák

3.2.1. Vektor

Azonos típusba tartozó elemek (például számok, szövegek, logikai értékek) listája. A vektorok sem oszlopban, sem sorban nincsenek rendezve.

Létrehozás:

- Konkatenáció - összefűzés (egymásba ágyazva is)
  - Példa: `a <- 5; b <- c(a,4); d <- c(4,5,c(b,6))`
- `rep` utasítással: ennek változatai: times, length, vagy each
  - Példa: `a <- rep(4,times=5); b <- rep(c(4,5),length.out=7);`
    `d <- rep(c(5,6),each=4)`
- `seq`: `length` vagy `by` megadásával
  - Példa: `a <- seq(1,30,by = 6); b <- seq(1,30,length.out = 6)`
- kettőspont: speciális utasítás (egészekből álló intervallumot hoz létre)
  - Példa: `a <- 30:47` pontosan ugyanaz, mint: `b <- seq(30,47,by=1)`
- egyéb: pl. beolvasva külső forrásból, vagy véletlenszám-generálással (ezeket később tárgyaljuk)

Tulajdonságok:

- Az elemek típusa és a vektor hossza, a `class`, illetve a `length` függvénnyekkel kérdezhetők le.
- Minden vektornak beállítható (alapértelmezetten nincs) egy `names` nevű attribútum, ez megjelenik többek között a vektor kiíratásakor a konzolon, vagy fájlba való mentésnél.
  - Példa: `v <- 1:4; names(v) <- c("A","B","C","D"); v;`

Algebrai operációk:

Az alapműveletek koordinátánként értelmezettek, a rövidebb vektort ciklikusan felhasználva. Ha a rövidebb vektor nem fogy el (azaz, hossza nem osztója a hosszabbik vektor hosszának), akkor warning keletkezik, nem hiba. Próbáljuk ki: `a <- 2:5; b <- c(1,2); d <- 4:6; z <- a + b; w <- b * d;`

Függvények:

A legtöbb R-be beépített függvény (`sin, cos, log`) koordinátánként értelmes, és így a legtöbb saját függvényünk is, ha csak ilyeneket használnak. Vannak olyan függvények is, amik ezt elrontják, pl. a `max, min` függvény összeolvassa az összes koordinátát, de ezeknek van vektorizált változata: `pmax, pmin`. Ha azonban bonyolultabb függvényt írunk, ami például `if`-et tartalmaz, ami egy warning mellett ignorálja a vektort, és csak az első koordinátát használja, akkor ezzel vektort megetetni életveszélyes, mert legjobb esetben elromlik, rosszabb esetben lefut, és teljesen marhaságokat csinál. Ezen segít a `Vectorize` függvény, mely egy új függvényt hoz létre az eredetiből, ami már vektorokon is értelmes. Példák:

```
x <- seq(-5*pi,5*pi,length.out= 1000);y <- sin(x);plot(x,y)
z <- max(y,0); plot(y,z) - ez hibát okoz
w <- pmax(y,0);plot(x,w)
```

Indexelés:

Indexelésre az egyszeres és dupla szögletes zárójel, az `[ ]` és `[[ ]]` operátor szolgál. A két operátor között lényeges különbség van, azonban ez vektorok esetén nem jön elő, ezért erre majd a listák kapcsán térünk ki. Vektorok esetén annyit elég megjegyezni, hogy a [[ ]] operátor a vektor egyetlen elemének kiválasztására alkalmas, és ha az adott elem nem létezik, akkor hibát generál, míg a [ ] operátor egynél hosszabb indexekből álló vektor esetén is működik, és „értelmetlen” indexelés esetén `NA`-val tér vissza, de nem generál error-t vagy warning-ot. Példaként legyen `a <- 5:10;` Ekkor mind `a[1]`, mind `a[[1]]` a vektor első elemét jelenti, ami 5 (pontosabban az 5-öt tartalmazó 1 hosszúságú vektor). Azonban `a[2:4]` egyenlő a `c(6,7,8)` vektorral, míg `a[[2:4]]` hibás kód. A másik különbség, hogy `a[7]` értéke: `NA`, míg az `a[[7]]` kifejezés `out of bonds` hibát generál. Az [[ ]] operátort nem szokták vektorok esetében alkalmazni, ha esetleg mégis, akkor kétféle indexet értelmes átadni neki:

- Egy természetes számot 1 és a vektor hossza között: ekkor értelemszerűen kiválasztja az annyiadik elemet. Ha a szám kisebb, mint 1, vagy nagyobb, mint a vektor hossza, akkor hiba keletkezik. R-ben a vektorok 1-től nem 0-tól indexelődnek.
- Egy string-et, azaz szöveget: ezt összeveti a vektor names attribútumával, és ha van egyezés, akkor a neki megfelelő elemet választja ki, egyébként hibát generál. Példa: `v[["A"]]; v[["E"]];`

A [ ] operátornak indexként általában a következő objektumok valamelyikét adjuk át:

- Egy természetes számot 1 és a vektor hossza között – ez értelemszerűen működik

```
a <- 5:10
a[2]
6
```

- Egy természetes számot, ami nagyobb, mint a vektor hossza – hiba nélkül NA-t ad vissza

```
a <- 5:10
a[10]
NA
```

- Nullát: ez nem okoz hibát, egy nulla hosszú változót ad vissza, nem választ ki semmit

```
a <- 5:10
a[0]
integer(0)
```

- NA-t: ilyenkor hiba nélkül NA -t választ ki a vektor *összes eleme* helyett.

```
a <- 5:10
a[NA]
[1] NA NA NA NA NA NA
```

- Negatív számot: az index abszolútértékének megfelelő indexű elemet kihagyja a vektorból

```
a<-5:10
a[-2]
5 7 8 9 10
```

- Egy természetes számokból és nullákból, illetve NA-kból álló bármilyen hosszú sorozatot (vektort): ilyenkor a nullákat ignorálja, a vektor hosszánál nagyobb indexekre és NA -kra NA -t ad vissza, az 1 és a vektor hossza közötti indexekre, a megfelelő indexű elemet választja ki, akár ismétléssel. A visszaadott vektor hossza az indexben lévő nem nulla elemek száma.

```
a <- 5:10
a[c(1,0,NA)]
5 NA
a[c(10,0,1,NA)]
NA 5 NA
```

- Egy negatív számokból és nullákból álló sorozatot (NA-t nem tartalmazhat, az hibát okoz): ilyenkor a nullákat ignorálja, a negatív indexek abszolútértékének megfelelő indexű elemeket kihagyja a vektorból, ha van olyan indexű elem. A vektor hosszánál abszolútértékben nagyobb számokat figyelmen kívül hagyja.

```
a <- 5:10
a[c(-10,0,-1)]
6 7 8 9 10
```

- Negatív és pozitív számokat, illetve negatív számokat és NA-kat egyszerre tartalmazó indexvektor hibát okoz, negatív számok csak nullákkal keverhetők.

```
a <- 5:10
a[c(-10,0,-1, 1, NA)]
Error in a[c(-10, 0, -1, 1, NA)] :
only 0's may be mixed with negative subscripts
```

- Ha törteket tartalmazó vektort adunk át indexnek, akkor csonkolja őket (nem a matematika szabályai szerint kerekíti) és egész számként értelmezi.

```
a<-5:10
a[1/2]
integer(0)
a[5/4]
5
```

- Logikai vektort: Ez a TRUE és FALSE (használható a T és F rövidítés) logikai konstansokat tartalmazó vektor. Ha a vektor hossza legalább akkora, mint az indexek száma, akkor az indexvektor ciklikusan kiegészül (ez esetben warning nélkül), és kiválasztja a vektorból a TRUE helyeken lévő elemeket (a visszaadott vektor hossza megegyezik az indexvektorban lévő TRUE-k számával). Ha az indexvektor hosszabb, mint az adott vektor, akkor a vektor hosszát meghaladó indexekre a FALSE-t ignorálja, a TRUE-kra pedig NA-t ad vissza.

```
a<-5:10
a[T]
5 6 7 8 9 10
a[F]
integer(0)
```

```
7>a[10]

NA

7>a[1]

TRUE
```

- A logikai vektorok NA-kal keverhetőek, ez esetben minden NA -ra kiválaszt egy NA -t, a többi indexet az előző pontban leírtak szerint használja.

```
a<-5:10

a[c(T,NA)]

5 NA 7 NA 9 NA
```

- A 0-1 vektorokat nem értelmezi logikai vektorként, hanem egész számoknak megfelelően, a nullákat ignorálja, és minden 1 indexhez kiválasztja a vektor első elemét.

```
a<-5:10

a[c(T,F)]

5 7 9

a[c(0,1)]

5
```

- Ha a logikai értékeket egész számokkal keverjük, akkor a logikai értékek konvertálódnak egész számokká (figyelmeztetés nélkül), így például TRUE és -2 keveredése hibát okoz.

```
a<-5:10

a[T*2]

6

a[F+1]

5
```

- Tetszőleges hosszúságú character típusú (sztringekből álló) vektor. Ezt a names argumentummal veti össze a program, és ha szerepel benne az adott sztring, akkor kiválasztja a neki megfelelő elemet, ha nem, akkor NA-t választ ki, hiba és warning nélkül.

```
v <- 1:4 names(v) <- c("A","B","C","D") v["B"] B 2
```

- Üres, azaz teljesen hiányzik. Ez a vektor összes elemének kiválasztását jelenti egy új változóba, amelybe így nem kerülnek át az attribútumok.

  ```
  a<-5:10
  a[]
  5 6 7 8 9 10
  ```

- Bármilyen nulla hosszúságú vektor (például egy sohasem teljesülő feltétel kiértékelésekor keletkező logikai vektor, vagy NULL objektum). Ez üres vektort ad vissza.

  ```
  a<-5:10
  a[a>100]
  integer(0)
  ```

Megjegyzés: logikai vektor létrehozható relációs operátorokkal, például, ha u és v két vektor, akkor a v <= u kifejezés egy olyan logikai vektort ad vissza, ahol TRUE érték szerepel azoknál a koordinátáknál, ahol a reláció teljesül, és FALSE ahol nem (az u és v vektor ekkor is ciklikusan használódik fel).

3.2.2. Mátrix

Azonos típusú elemekből álló kétdimenziós tömb.

Létrehozás:

- Elvileg az array utasítással, mint tömböt, ha a dim változó hossza 2, de ezt ritkán használjuk.
- A matrix utasítással. Például a

  ```
  A <- matrix(data = 1:12, nrow = 4, ncol = 3, byrow = FALSE, dimnames =
                                        NULL)
  ```

  utasítás létrehozza az A 4×3-as mátrixot, oszloponként az 1-től 12-ig terjedő egész számokból. Az összes paraméter opcionális. Az adatokat tartalmazó data változó bármilyen vektorként értelmezhető kifejezés lehet, amelynek a vektoroknál látott ciklikus felhasználása megengedett.

  Megadható a sorok és oszlopok száma, de ha elhagyjuk, a függvény az átadott adatok mennyiségéből megpróbálja "kitalálni". Ha a byrow paramétert TRUE-ra állítjuk, akkor értelemszerűen soronként jön létre a mátrix.

Tulajdonságok:

A mátrixokról lekérdezhető legalapvetőbb tulajdonságok a mátrix elemeinek száma (`length`), a sorok száma (`nrow`) és az oszlopok száma (`ncol`). Ezeken kívül létezik egy `dimnames` nevű attribútum, ami egy kételemű lista a sor és oszlopnevekkel, ez a létrehozásnál, illetve utólag is beállítható, akár a `dimnames`, akár külön-külön a `rownames` és `colnames` függvényekkel.

Algebrai operációk:

A négy alapművelet mátrixokra a vektorokhoz hasonlóan elemenként működik, de ezeket elvégezni csak azonos méretű mátrixokra hajlandó az R, itt nincs ciklikus felhasználása a változóknak, nem megfelelő méretű mátrixok esetén hibaüzenettel leáll a kód. Mátrixokra értelmezhető egy speciális `%*%` -kal jelölt művelet, ez a mátrixszorzás. Ez is csak összeszorozható mátrixok esetén működik, egyébként hibaüzenettel leáll.

Függvények:

Az elemi függvények (`sin, cos, log`, stb.) szó szerint ugyanúgy elemenként működnek, mint vektorokra. Mátrixokra vonatkozó speciális függvények a `cbind` és `rbind`, ezek összeragasztják a mátrixokat egymás mellé, illetve egymás alá, ha ugyanannyi soruk, illetve oszlopuk van (ha nem, akkor hibaüzenettel leállnak, nincs ciklikus felhasználása a mátrix elemeinek). Kiválasztható a mátrix főátlója a `diag` függvénnyel. Ezen kívül mátrixokat lehet transzponálni a `t` függvénnyel, és invertálni a `solve` függvénnyel. Négyzetes mátrix sajátértékeit és sajátvektorait az `eigen` függvénnyel, determinánsát a `det` függvénnyel kaphatjuk meg.

Indexelés:

A mátrix egyes elemeinek eléréséhez két változót kell átadni a `[ ]`, illetve a `[[ ]]` operátornak, vesszővel elválasztva. Tehát például az A mátrix i-edik sorának j-edik elemét az `A[i,j]`, vagy az `A[[i,j]]` szimbólummal lehet elérni. Az indexelés csak akkor működik, ha létező elemre mutat, azaz, ha `1 <= i <= nrow(A), 1 <= j <= ncol(A)`. Mátrixok esetén az egyszeres `[ ]` zárójel is „out of bonds” hibát okoz, ha valamelyik index kilóg a megengedett tartományból. A mátrix indexelése általában `A[ind1,ind2]` alakban történik. Ha csak egyetlen indexet használunk, akkor a mátrix oszloponként vektorrá konvertálódik és annak megfelelően indexelődik (kivéve, ha az átadott index nulla hosszú, ez esetben a teljes mátrix kerül kiválasztásra). Az `A[ind1,ind2]` alakú indexelés legfontosabb esetei, ha ind1 és ind2 egymástól függetlenül az alábbi alakú:

- Pozitív egészekből álló vektor, melynek elemei nem nagyobbak, mint a sorok, illetve oszlopok száma. Ha mindkét indexvektor hossza nagyobb, mint egy, akkor a megfelelő indexű sorokból és oszlopokból álló részmátrix kerül kiválasztásra.

- Ha ind1 vagy ind2 egyetlen indexet tartalmaz (1 hosszúságú vektor), azaz egyetlen sor vagy oszlop bizonyos elemeit választjuk ki, akkor az alapértelmezett beállítás szerint a kiválasztott "részmátrixot" vektorrá konvertálva kapjuk vissza (elveszíti mátrix jellegét). Ezután az `nrow`, és `ncol` függvények, illetve a dupla indexek többé nem működnek rá. Ez számtalan buta hibát okozhat egy kód futása során. Erre a problémára a „dimenziócsökkentés" részben részletesen kitérünk.

- Ha olyan indexet adunk meg, ami nagyobb, mint a felhasználható legnagyobb (sorok, illetve oszlopok száma) akkor nem választódnak ki `NA`-k, hanem "subscript out of bonds" kivétel keletkezik, és a programfuttatás leáll.

- Logikai értékekből álló vektor, mely meghatározza, mely oszlopok és sorok választandóak ki a mátrixból. Ha valamelyik indexvektor túl rövid, akkor ciklikusan újrahasználódik, ha túl hosszú, akkor viszont hiba keletkezik.

- Negatív egész számokból álló vektor, ez esetben a negatív indexeknek megfelelő sorok, illetve oszlopok törlésével kapott mátrix választódik ki. Ha valamelyik index abszolútértékben nagyobb, mint a sorok, illetve oszlopok száma, akkor nem keletkezik "out of bonds" kivétel, csak elhagyjuk az egyébként sem létező sort, illetve oszlopot.

- Karakterekből álló vektorok, ezek a `rownames` és `colnames` attribútumokkal kerülnek összehasonlításra. Nem létező címkék esetén nem `NA`-t kapunk, hanem hiba keletkezik.

- Ha valamelyik index hiányzik (de a "," ki van téve a `[ , ]`-ben), az az összes sor, illetve oszlop kiválasztását eredményezi.

- `NA`-t tartalmazó index esetén egy teljes `NA`-t tartalmazó sor, illetve oszlop kerül kiválasztásra.

3.2.3. Tömbök

A mátrixok általánosításaként létrejövő „n-dimenziós téglatest" alakú tartományok. Precízebben, az n-dimenziós tömb olyan vektor, melynek `dim` attribútuma egy n-hosszú vektor. Létrehozás: A tömb az `array` utasítással jön létre. A függvénynek át kell adni az adatokat és a `dim` változót, egy pozitív egész számokból álló vektort, ami megadja, hogy melyik dimenzióban meddig terjed a koordináták száma. Noha a tömb gyakorlatilag csak egy vektor, ami tartalmazza a `dim` attribútumot, szemléletesen úgy lehet rá gondolni, hogy az az eredeti adatokat egy 'n-dimenziós mártrix'-ba ('téglába', 'kockába') rendezi, mindig a legelső (legbaloldalibb) koordinátát növelve legelőször. Tehát három dimenzióban először feltölt egy `dim[1]` hosszúságú vektort, ha az betelt, akkor oszloponként egy `dim[1]` $\times$ `dim[2]` méretű mátrixok, ha az is betelt, akkor a mátrix „mögé" ragaszt egy ugyanakkora mátrixot, és ha az is betelt, akkor egy újabb mátrixot az előző „mögé".

Tulajdonságok:

A tömb egy vektor egy dim és egy opcionális dimnames attribútummal, mely a mátrix sor- és oszlopneveinek általánosításaként az egyes dimenziók mentén megadott feliratokat tartalmazza.

Algebrai operációk:

A négy alapművelet ugyanúgy működik, mint mátrixok esetén, megegyező méretű tömbök adhatóak, szorozhatóak stb. össze, egyéb értelmes műveletet tömbök esetén nehéz elképzelni.

Függvények

A szokásos (vektorokra komponensenként alkalmazható) elemi, egyváltozós függvények, tömbökre elemenként alkalmazhatóak, és ugyanolyan méretű tömböt hoznak létre, mint az eredeti, a kiszámolt értékekkel.

Indexelés

A `[ ]` és `[[ ]]` operátorok ugyanúgy alkalmazhatóak, mint mátrixokra, értelemszerűen annyi koordinátát kell átadni, ahány dimenziós a tömb, vagy átadható egyetlen index, ez esetben a tömb a mátrixhoz hasonlóan vektorrá konvertálódik és annak megfelelően indexelődik.

3.2.4. Mátrixokkal és vektorokkal vegyesen végezhető műveletek

A mátrixok és vektorok R-ben lényegében teljesen egyforma objektumok abban az értelemben, hogy valójában mindegyik vektor, emellett mindkét adatstruktúra teljesen homogén, vagyis csak ugyanolyan típusú (`integer, numeric, logical, character`) adatot tartalmazhat. Ennek köszönhetően bizonyos függvények és műveletek vegyesen is alkalmazhatóak rájuk, vagyis akkor is működnek, ha egyik argumentumuk egy mátrix, másik pedig egy vektor. A legfontosabb ilyen példa a mátrixszorzás, ami „mátrix %*% mátrix” megadással csak akkor működik, ha a mátrixok összeszorozhatóak (vagyis annyi oszlopa van a bal oldali mátrixnak, mint ahány sora a jobboldalinak). Azonban „mátrix %*% vektor”, illetve „vektor %*% matrix” alakban megadva, a vektor automatikusan a megfelelő sor, illetve oszlopvektorrá konvertálódik, amennyiben megfelelő a hossza. Egy négyzetes mátrix tehát egy megfelelő hosszúságú vektorral mindkét irányból összeszorozható, transzponálás nélkül. Hasonlóan a „vektor %*% vektor” alakban megadva, a baloldali vektor sorvektorként, a jobboldali oszlopvektorként viselkedik, és megkapjuk a skaláris szorzatukat. A másik fontos példa: a `cbind`, illetve `rbind` függvény. `cbind(matrix,matrix)` alakban csak akkor működik, ha a két mátrixnak ugyanannyi sora van. `cbind(matrix,vektor)` alakban azonban a vektor ciklikusan felhasználódik, illetve csonkolódik, ha túl hosszú. Ebben az esetben természetesen az eredmény mátrixnak mindig eggyel több oszlopa lesz, mint az eredetinek.

3.2.5. Dimenziócsökkentés

Legyen `A` egy mátrix, melynek egy algoritmus kiválasztja valahány oszlopát, de változó, hogy hányat, és az így kapott részmátrixszal dolgozik tovább. Például a mátrix oszlopai különböző változók megfigyeléseit tartalmazzák, és ezekre szeretnénk mindenféle kombinációban regressziós modelleket illeszteni. A kód tökéletesen működik, mindaddig, amíg egyszer csak elő nem fordul, hogy csak *egyetlen* oszlopot választunk ki a mátrixból. Ekkor ugyanis a kiválasztott részmátrix vektorrá konvertálódik, és értelmetlenné válik az elemeire való hivatkozásban kettős indexeket használni, valamint hibát okoz, ha a sorainak vagy oszlopainak számát próbáljuk lekérdezni. A futó kódban tehát külön kéne vizsgálni azt az esetet, ha a kiválasztott sorok vagy oszlopok száma 1, és külön az összes többit. Egy nagyon egyszerű példával legyen

```
i <- 2; A <- matrix(1:16, 4,4); B <- A[1:3,3:(3+i-1)]; B[nrow(B),1]
```

Ez a kód hibátlanul működik. Azonban, ha úgy alakul, hogy csak egyetlen oszlopot akarunk kiválasztani, akkor „incorrect number of dimensions” hibát kapunk:

```
i <- 1; A <- matrix(1:16, 4,4); B <- A[1:3,3:(3+i-1)]; B[nrow(B),1]
```

Ennek oka, hogy a `[ ]` operátor igyekszik az általa visszaadott változót a lehető legkisebb dimenziójúra csökkenteni. Ez praktikusan azt jelenti, hogy a `dim` attribútumban lévő 1-eseket kihagyja, és ha így a hossza 1-re csökken, akkor teljesen törli. Így az egy oszlopból álló mátrixok vektorrá konvertálódnak.

Ez az alapértelmezett viselkedés szerencsére felülírható. Ha az indexelésnél a `[ ]` operátor drop opcionális paraméterét `FALSE`-ra állítjuk, akkor a dimenziócsökkentést az R nem végzi el. A következő kód már hibátlanul működik:

```
i <-1; A<-matrix(1:16,4,4); B <- A[1:3,3:(3+i-1), drop = FALSE]; B[nrow(B),1]
```

Ugyanezt a B-ből való elem kiválasztásánál is megtehetjük, ha `B[nrow(B),1,drop = FALSE]` –t írunk, ekkor a visszaadott érték egy 1x1-es mátrix lesz.

3.3. Elemi adatstruktúrák attribútumai

Hogy jobban megértsük az adattárolás és attribútomok kezelésének logikáját, tekintsük az alábbi példát. Írjuk be az R konzolba, hogy `A <- 1:12`. Ezzel létrehoztunk egy 12 elemű egész számokból álló vektort. A `class(A)` beírásával erről meg is győződhetünk, ezzel lekérdezzük a változó típusát. Adjuk most ki a `dim(A) <- c(2,6)` utasítást, majd irassuk ki a változót (írjuk be a konzolba a nevét). A változó most már kétszer hatos mátrix alakban jelenik meg a képernyőn, és a `class(A)` utasítás ismételt beírására már a `matrix` választ kapjuk. A mátrix, a vektor és a tömb valójában ugyanaz a struktúra, a köztük lévő különbséget csak a `dim` attribútum határozza meg. Ha beírjuk a konzolba, hogy `dim(A) <- c(2,3,3)` akkor azt láthatjuk, hogy ugyanez a változó most háromdimenziós tömbbé változott.

Az attribútumok határozzák meg az adatokkal végezhető műveleteket, például a mártixszorzás

elvégzésére az R akkor hajlandó, ha a `dim` attribútumok kompatibilis mátrixokat jelölnek. Utolsó példaként definiáljuk a következő változókat:

```
a <- -5:6
b <- rep(0,12)
```

A két vektor koordinátánként vett minimumát a `pmin` függvénnyel számolhatjuk ki. A `pmin(a,b)`, és a `pmin(b,a)` függvényhívás ugyanazt a vektort adja, a korrekt eredményt. Hozzunk létre most a két vektorból két különböző méretű mátrixot:

```
A<-matrix(a,3,4)
B <- matrix(b,4,3)
```

Ekkor a `pmin(A,B)` mátrix 3×4-es, míg a `pmin(B,A)` 4×3-as, a két mátrix elemei azonban oszloponként felsorolva megegyeznek. A `pmin` függvény kiszámolja a koordinátánkénti minimumokat a két mátrix adataiból, majd az eredmény `names` és `dim` attribútumát kimásolja az *első* változóból, így az eredmény mátrix méretét csupán ez határozza meg. Ez, a természetes szimmetriát megtörő jelenség R-ben nem ritka. Hogy a történet még cifrább legyen, ha a két mátrixnak nem ugyanannyi eleme van, akkor mindkettő vektorrá konvertálódik, és a rövidebb ciklikusan felhasználódik, az eredmény pedig egy vektor, tehát ebben az esetben egyik változóból sem másolódnak az attribútumok.

4. fejezet

Összetett adatstruktúrák

A továbbiakban a teljesség igénye nélkül megismerkedünk néhány olyan - összetettebb változótípussal, melyek az adatelemzés és különböző statisztikai modellek kapcsán leggyakrabban használatosak.

4.1. Lista

A lista olyan összetett adatstruktúra, ami az eddigieknél bonyolultabb objektumok leírására is alkalmas, ugyanis, nemcsak hogy tartalmazhat különböző típusú elemeket, hanem tetszőleges R-beli objektumokat tárolhat, például vektorokat, mátrixokat vagy akár újabb listákat, bármilyen mélységig egymásba ágyazva, sőt függvényeket is. Az alábbi kód például egy kételemű (igen, kételemű) listát hoz létre:

```
L <- list(x = 1:5, A = matrix(1:20,4,5))
```

Létrehozás:

Listát a `list` függvénnyel lehet létrehozni, illetve a bonyolultabb eljárások, például regressziós modellek outputjaként keletkezik. A `list` függvény futtatásakor át lehet adni ”kulcs = érték” formátumban a felhasználandó elemek listáját. Az = jel előtt string-ek állnak idézőjelben vagy anélkül, ezek alkotják majd a kulcsokat, melyek a lista `names` attribútumába kerülnek. A kulcsok megadása nem kötelező, felsorolhatjuk a listát alkotó objektumokat ezek nélkül is, vesszővel elválasztva.

Tulajdonságok:

A lista annyira általános struktúra, hogy mindössze két dolgot lehet tudni róla általánosságban, az egyik a hossza, melyet itt is a `length` függvény ad meg, a másik a típusa, ami `list`. Opcionálisan lehet egy `names` nevű attribútuma, ha megadtuk, mely a lista hosszával megegyező hosszúságú `character` típusú vektor. A nevek nem feltétlenül kell, hogy egyediek legyenek, de erősen ajánlott.

Algebrai operációk:

Listákkal algebrai műveletek nem végezhetőek, az ilyen kód azonnal hibára fut.

Függvények:

Listákra az R numerikus függvényei nem alkalmazhatóak. Vannak függvények, melyek paraméterként listákat is elfogadnak (pl. `plot`), ezek a listát, mint összetett objektumként és nem vektorként kezelik. Ezzel egyelőre nem foglalkozunk.

Indexelés:

A lista indexelésére háromféle operátor is van:

- Az egyszeres `[ ]` zárójel a *subset* (”részhalmaz”) operátor, melynek vektorokat is átadhatunk indexként. Ennek visszatérési értéke mindig egy lista, mely a kiválasztott elemeket tartalmazza. Tehát például `L <- list(1,2,3:5)` esetén `L[2:4]` egy lista, melynek elemei a 2, a 3:5 vektor, valamint a `NULL` speciális null-objektum, mely valamilyen értelemben az `NA` logikai érték megfelelője, amely a nem létező index miatt lett „kiválasztva” L-ből. Az `[ ]` operátornak a vektoroknál megszokotthoz hasonlóan átadhatunk egész számokból, logikai értékekből, illetve string-ekből álló vektorokat és az indexelés a vektoroknál látotthoz hasonlóan működik.
- Az `[[ ]]` operátor a lista egyetlen elemének kiválasztására szolgál, ennek tehát csak egyetlen stringet vagy egész számot adhatunk át indexként. A stringet a lista `names` nevű argumentumában keresi, az egész számra pedig a lista annyiadik elemét veszi ki.
- Listák indexelésének harmadik módja a $ operátor. Ez csak a names attribútummal rendelkező listáknál használható. Ha az L nevű listának van „valami” nevű eleme, akkor az `L$valami` ezt választja ki. Ez lényegében ekvivalens az `L[[„valami"]]` kifejezéssel. A kettő közti fő különbség az, hogy a `[[ ]]` operátornak a belsejében lehet függvényhívás, vagy valamilyen kiértékelendő kifejezés, míg a `$` operátor használata esetén konkréten be kell írni a megfelelő nevet. A másik különbség, hogy a `$` operátor NULL objektumot ad vissza, nemlétező név esetén a `[[]]` viszont hibával leáll.

Fontos hangsúlyozni a `[ ]` és `[[ ]]` operátorok közti különbséget. Hogy jobban megértsük, legyen a `fiok` nevű változó egy lista, aminek három eleme van, zokni, szemüveg és telefon.

```
fiok <- list("zokni", "szemuveg", "telefon")
```

A fiókból kivenni a telefont a `[[ ]]` operátorral lehet: `tel <- fiok[[3]]`. Ekkor `class(tel)` `character` lesz.

Ezzel szemben a ”részhalmaz” operátor egy ugyanolyan szintű objektumot ad vissza, mint az eredeti, azaz `fiok[1]` létrehoz egy fiókot (listát), amelyben csak zokni van, de ezzel a zoknit nem vettük ki a fiókból. A `class(fiok[1])` továbbra is `list`.

4.2. Data.frame

A `data.frame` definíció szerint olyan lista, melynek elemei azonos hosszúságú (de nem feltétlenül azonos típusú) elemi vektorok, és a `class` attribútuma „data.frame”. Ez gyakorlatilag egy téglalap alakú adattábla, melynek oszlopai általában változókat, sorai megfigyeléseket tartalmaznak. A `data.frame` az R sztenderd adattárolásra használt objektuma, mely leggyakrabban külső fájlból történő adatbeolvasással jön létre és a legtöbb statisztikai modell ebben a formátumban „szereti” megkapni a felhasználandó adatokat.

Létrehozás:

A `data.frame` általában külső adattáblák beolvasásakor jön létre, amit leggyakrabban a `read.table` függvénnyel végzünk. A másik gyakori eset, hogy egy mátrixot alakítunk `data.frame`-é például az `as.data.frame` konverziós függvénnyel. Emellett lehetséges, bár nem szokás `data.frame`-et létrehozni a `data.frame` függvénnyel, lényegében csak az adatokat tartalmazó vektort/vektorokat kell átadni, de beállítható töménytelen opcionális paraméter, például a fejlécek vagy az adatok típusa.

Tulajdonságok:

A `data.frame` kombinálja a mátrixok és listák tulajdonságait, tehát értelmes rá például az `nrow, ncol` függvény, illetve a `length`, de ez utóbbi nem az elemek, hanem az oszlopok számát adja meg (tehát ugyanaz, mint az `ncol`), van `dimnames, rownames, colnames` és egy `names` attribútuma, de a két utóbbi megegyezik. Class attribútuma `"data.frame"`, de ennek ellenére létezik a `dim` attribútuma is, és az `is.matrix` függvény `TRUE`-t ad vissza `data.frame`-ekre (és az `is.list` is).

Algebrai operációk:

`Data.frame`-mel elvileg végezhetők algebrai műveletek, tehát nem okoz alapértelmezetten hibát, de gyakran vezet értelmetlen művelet alkalmazásához. Ha például meg akarunk szorozni egy `data.frame`-et kettővel, akkor ezt a műveletet úgy értelmezi, hogy megpróbál minden változót, azaz minden oszlopot kettővel megszorozni. Amennyiben az adattábla tartalmaz nem numerikus adatot, akkor ez a művelet értelmetlen, és a kód hibára fog futni. `Data.frame`-ekkel mátrixszorzás nem végezhető, ehhez mátrixszá kell konvertálni az `as.matrix` függvénnyel.

Függvények:

A `data.frame`-ekre a mátrixokhoz hasonlóan (elemenként) működnek az R azon függvényei, melyek vektorokon értelmesek, de az alkalmazásuk ugyanúgy, mint az alapműveleteknél hibára fut, ha csak egyetlen oszlopuk is nem megfelelő adattípusú. A `data.frame`-ekre a mátrixokhoz hasonlóan működik az `rbind` függvény, azonban a `cbind` nem a pozíciójukat tekintve egymás alatti oszlopokat, hanem az azonos fejlécű oszlopokat „ragasztja össze", és ha a fejlécek nem stimmelnek, akkor hibára fut.

Indexelés

A `data.frame`-k indexelésekor mind a „mátrixos", mind a „listás" szintaktika használható. A `$` operátor ugyanúgy működik, mint listáknál, kiválasztja a megfelelő nevű (jelen esetben fejlécű) változót, mely ilyenkor egy vektor lesz, ami ennek megfelelően tovább indexelhető. Tehát pl. `d$valami[25]`, a „valami" nevű változó (valami fejlécű oszlop) 25.-dik eleme. A mátrixos `[ ]` és `[[ ]]` operátor ugyanúgy működik, mint mátrixokra, ha két indexet adunk át. Ha viszont csak egyet, akkor nem, mert ez esetben nem egyetlen elemet választ ki, hanem egy egész oszlopot (akár fejlécnevet, akár egész számot adunk meg).

4.3. Data.table

A `data.table` az `install.packages("data.table")` csomag telepítése után elérhető adatkezelési/ adattárolási formátum, amely a data.frame-hez képest sokkal jobban kezeli a nagyméretű adatbázisokat, és az adatbázis-lekérdezéseket. A `class` attribútuma egyszerre `"data.table"` és `"data.frame"`.

Létrehozás:

A `library(data.table)` csomag betöltése után hozhatjuk létre `data.table()` utasítással. A `setDT()` függvény meghívására (egy adott adattáblára például `adattabla<-setDT(adattabla)`) az adattábla `data.table` formátumú lesz.

Tulajdonságok:

Minden `data.table` egyszerre `data.frame` is, így minden olyan tulajdonsággal rendelkezik, mint a `data.frame`. De a `data.frame` struktúráján túlmutat, hiszen a szerkezete `DT[i, j, by]`, azaz alkalmas az adatok oszlopok szerinti csoportosítására is.

Algebrai operációk:

`Data.table`-el (hasonlóan a `data.frame`-hez) elvileg végezhetők algebrai műveletek, de hasonló hibákhoz vezethet az alkalmazásuk.

Függvények:

A `data.table`-ökre teljesen azonosak mint a `data.frame`-k esetében.

Indexelés

A `data.table` indexelésekor mind a „mátrixos”, mind a „listás” szintaktika használható, de nem ez a jellemző használata, hanem a speciálisan ezen csomagban található indexelések a jellemzőek.

Data.frame és data.table közti legfontosabb különbségek

Data.frame létrehozása

```
d<-data.frame(id=1:4,company=c("OTP","MKB","KH","OTP"), value=c(1/3,2/3,4/3,3))
```

Data.table létrehozása:

```
library(data.table)
DT <-data.table(id=1:4,company=c("OTP","MKB","KH","OTP"),
   value = c(1/3,2/3,4/3,3))
```

Művelet	**data.frame**	**data.table**
Oszlop kiválasztás	`d$value`	`DT[,value]`
Több oszlop kiválasztás	`d[,c("value","company"]`	`DT[,(value,company)]`
Művelet elvégzése	`mean(d$value)`	`DT[,mean(value)]`
Csoportosítás oszloponként	`aggregate(formula = value company, data = d, FUN = mean)`	`DT[,mean(value),by = company]`
Új oszlop beszúrása	`d$ujvaltozo <- d$value + 1`	`DT[,ujvaltozo := value + 1]`
Sorokra vonatkozó művelet	`apply(d[,c("company", "value")], 1,FUN = function(x) paste(x,collapse = " - "))`	`DT[,paste(company, value,sep=" - ")]`

Egyéb példák a data.table használatára:

https://s3.amazonaws.com/assets.datacamp.com/blog_assets/datatable_Cheat_Sheet_R.pdf

5. fejezet

Adatstruktúrákhoz kapcsolódó gyakorlófeladatok

Az alábbi feladatok főként mátrixok és vektorok legelemibb tulajdonságainak felhasználásával oldhatóak meg. Semmilyen nem triviális statisztikai vagy lineáris algebrai alkalmazást nem tartalmaznak, fő céljuk, hogy a más nyelvekben általában ciklusokkal megoldható feladatokat az R-ben sokkal hatékonyabb vektorműveletekkel oldjuk meg. A fejezet végén feltüntetünk egy vagy több lehetséges megoldást. A vektorok és mátrixok megkülönböztetésére a vektorokat ()-be, a mátrixokat pedig []-be tesszük.

5.1. Feladatok

1. FELADAT

   Definiáljuk és írassuk ki a következő vektorokat!

   `a = 1 b = -1 c = (1,2) d = (1,2,3) u = (2,5,8,...,200)`

   `w = (100, 99, 98,...51) z = (-100,-99,-98,...,99,100)`

   Számoljuk ki a következő összegeket: `a+b a+c b+c w+c`!

   Mikor kapunk warning-ot?

   Fűzzük egymás után a c és d vektorokat!

2. FELADAT

   Hozzuk létre a következő vektort: `x <- ("OTP", 5, 2/3)`!

   Mi történik, ha megpróbáljuk megszorozni a vektort 3-mal? Mi történik, ha megpróbáljuk megszorozni a vektor második elemét 3-mal? Konvertáljuk számmá a vektort és szorozzuk meg 3-

mal! Mi történik a vektorban lévő szöveggel? Konvertáljuk a vektort egész számmá és szorozzuk meg 3-mal! Mi történik a vektorban lévő törttel?

Hozzunk létre egy listát ugyanezen elemekkel! Próbáljuk megszorozni a listát 3-mal. Próbáljuk megszorozni külön-külön a lista elemeit 3-mal! Mit tapasztalunk?

3. FELADAT

Rakjunk össze az a, b változókból egy 1000 hosszúságú vektort, ami felváltva tartalmazza a +1 és -1 számokat!

4. FELADAT

Írj egy kódot, ami összeadja a 400-nál nem nagyobb négyzetszámokat!

5. FELADAT

Hozzuk létre a (-10,-9,...,-1,0,1,2,...,10) vektort! Írjunk kódot, ami kinullázza a vektor 5-nél nagyobb abszolútértékű elemeit!

6. FELADAT

Számítsuk ki minél pontosabban a négyzetszámok reciprokaiból álló végtelen sor összegét:

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} \cdots = \frac{\pi^2}{6}$$

7. FELADAT

Számítsuk ki minél jobb közelítéssel a következő végtelen összeget:

$$\sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n} = +\frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4^2} \cdots = \log(2)$$

8. FELADAT

Hozzuk létre a következő vektorokat és mátrixokat:

$$A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 3 & 1 \\ 1 & 1 & 4 \end{bmatrix} \quad u = (1,2,3) \quad v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad w = \begin{bmatrix} 1, & 2, & 3 \end{bmatrix} \quad B = \begin{bmatrix} 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{bmatrix} \quad z = (2,3,4)$$

Próbáljuk ki, mely műveleteket lehet elvégezni az alábbiak közül (sima * az elemenkénti szorzást, %*% a mátrixszorzást jelöli R-ben):

`v%*%A` `w%*%A` `A%*%v` `A%*%w` `A%*%u` `u%*%A` `u*A` `A*u` `A%*%B` `B%*%A` `u*z` `z*u` `u%*%z` `z%*%u`

9. FELADAT

Szorozzuk meg az alábbi A mátrixot a transzponáltjával, és számítsuk ki a szorzatban a főátló elemeinek összegét!

$$A = \begin{bmatrix} 1 & 10 & 8 \\ 4 & 1 & 10 \\ 8 & 4 & 1 \end{bmatrix}$$

10. FELADAT

Hozzunk létre egy 19×19-es mátrixot, ami sakktábla-szerűen tartalmazza a 0 és 1 számokat! Hozzunk létre ugyanígy egy 20×20-as mátrixot is! Ez miért nehezebb?

11. FELADAT

Szorozzuk össze az $x = (3,5,4,9)$ és $y = (6,3,8,2)$ vektorokat skalárisan (sor-oszlop)! Ezután szorozzuk őket össze diadikusan (oszlop-sor) és számoljuk ki a kapott mátrix elemeinek összegét!

12. FELADAT

Hozzuk létre az alábbi mátrixot:

$$\begin{bmatrix} 2 & 1 & 0 & 0 & \dots & 0 \\ 1 & 2 & 1 & 0 & \dots & 0 \\ 0 & 1 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \ddots & \ddots & 1 & \vdots \\ \vdots & \vdots & \ddots & 1 & 2 & 1 \\ 0 & 0 & 0 & \dots & 1 & 2 \end{bmatrix}$$

13. FELADAT

Hozzuk létre az alábbi úgynevezett Wandermonde mátrixot:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 4 & 8 & 16 & 32 & 64 \\ 3 & 9 & 27 & 81 & 243 & 729 \\ 4 & 16 & 64 & 256 & 1024 & 4096 \\ 5 & 25 & 125 & 625 & 3125 & 15625 \\ 6 & 36 & 216 & 1296 & 7776 & 46656 \end{bmatrix}$$

5.2. Megoldások

1. A vektorok megadhatóak többféleképpen, például:

   ```
   a <- 1; b <- -1; c <- c(1,2); c <- 1:2; d <- c(1,2,3); d <- 1:3
   ```

   ```
   u <- seq(2,200, by = 3); w <- seq(100,51, by =-1); w <- 100:51;
   ```

   ```
   z <- seq(-100,100, by =1); z <- -100:100
   ```

   A vektorok elemenkénti összeadása simán az `a+b a+c b+c` és `w+c` képletek beírásával történik.

   A rövidebb vektor ciklikus felhasználása mindig megtörténik, tehát egyik művelet sem okoz hibát, de ha a rövidebb vektor ”nem fér rá” valahányszor a hosszabbra, azaz az elemei ”nem fogynak el”, nem az utolsó elemével végződik az utolsó művelet, akkor warning jelenik meg a konzolban. Ez akkor történik, ha a rövidebb vektor hossza nem osztója a hosszabb vektor hosszának.

   A vektorok összefűzése a `c` nevű függvénnyel történik, egyszerűen át kell adni neki megfelelő sorrendben az összefűzendő vektorokat: `c(c,d)`. Vegyük észre, hogy itt `c` egy változó neve is, és egy függvény neve is, ami nem okoz problémát. A program tudja, hogy ha egy szimbólumot egy ”(” követ, akkor függvényt kell keresnie, ha nem, akkor változót.

2. A vektor elemei csak azonos adattípushoz tartozhatnak, így a különbözőek automatikusan valamilyen közös típussá konvertálódnak. Ez jelen esetben a ”legalacsonyabb” szintű character típus, mert ez nem jár adatvesztéssel. Az ilyen típussal azonban az aritmetikai műveleteket nem lehet elvégezni, azok hibára futnak. Ha a vektort számmá konvertáljuk (`as.numeric(x)`), akkor a számként nem értelmezhető ”OTP” szöveg a hiányzó értékek jelölésére szolgáló `NA` szimbólummá konvertálódik és a benne lévő adat elvész. Erre egy warning is figyelmeztet. Egész számmá konvertálás a törtek csonkolását és nem kerekítését eredményezi, vagyis a tizedesjegyek mind elvesznek, a $\frac{2}{3}$-ból például 0 lesz.

   A listát az `xl <- list("OTP",5,2/3)` utasítással hozhatjuk létre. Mivel a lista tartalmazhat különböző típusú elemeket, nincs típuskonverzió a lista létrehozásakor. A lista számokat tartalmazó elemei számok maradnak, így azokkal minden szokásos művelet elvégezhető. Magát a listát azonban nem lehet hárommal megszorozni akkor sem, ha csak számokat tartalmaz, mert listákon nincsenek elemenként értelmezett műveletek.

3. `rep(c(-1,1), length.out = 1000)` vagy `rep(c(-1,1), times = 500)`

4. Mivel $\sqrt{400} = 20$, az 1 és 20 közötti egész számok négyzeteinek összegét kell kiszámolni. Erre egy helyes kód például:

   ```
   sum((1:20)^2)
   ```

Érdemes rá odafigyelni, hogy a `sum(1:20^2)` kód nem jó, mert ez a műveletek elvégzési sorrendje miatt a `sum(1:(20^2))` kóddal egyenértékű.

5. `a <- -10:10; a[abs(a)>5] <- 0`

6. Számítsuk ki például a sorozat első 100 tagját, az már elég jó közelítés: `sum(1/(1:100)^2)` (a `sum(1/1:100^2)` képlet nem jó)!

7. A sorösszeget az első N tag összegével becsülhetjük így: `sum((-1)^(1:N+1))/1:n)`.

8. Az `A` mátrix legfájdalommentesebben talán a következő kóddal hozható létre: először létrehozzuk csupa 1-gyel, `A <- matrix(1,3,3)`, majd átírjuk a főátlót: `diag(A) <- 2:4`. Az `u` vektor többek között az `u <- 1:3` kóddal állítható elő. A `v` oszlopmátrix létrehozható a `v <- matrix(1:3, 3,1)` utasítással, vagy rövidebben a `v <- cbind(1:3)` kóddal is. A `w` sormátrix hasonlóan létrehozható a `w <- matrix(1:3, 1,3)` vagy `w <- rbind(1:3)` kóddal. A többi értelemszerűen: `B <- matrix(c(2:4,6:8), 3,2)`, `z <- 2:4`.

   A példában lévő műveletek négy kategóriába sorolhatóak.

   A mátrixszorzás mátrixok között akkor végezhető el, ha a mátrixok matematikai értelemben összeszorozhatóak, vagyis annyi oszlopa van a baloldalinak, ahány sora a jobboldalinak. Például a `v%*%A` mátrixszorzás nem értelmes, ahogy az `A%*%w` sem, az `A%*%v` viszont igen, és ennek eredménye egy 3×1-es mátrix.

   A mátrixszorzás mátrixok és vektorok között, illetve vektorok és vektorok között valamivel rugalmasabb. Ezek akkor működnek, ha a baloldali vektorokat sorvektornak, a jobboldali vektorokat pedig oszopvektornak tekintve a műveletek értelmessé válnak. Például az `A%*%w` szorzással ellentétben az `A%*%u` és az `u%*%A` is elvégezhető, mert `u` oszlopvektor és sorvektor szerepét is betöltheti.

   A * pontonkénti szorzás mátrixok között csak akkor végezhető el, ha a mátrixok azonos alakúak (tehát `A%*%v` például nem), és ekkor az eredmény is egy ugyanolyan mátrix.

   A pontonkénti szorzás mátrixok és vektorok között vegyesen (például `A*u`), illetve vektorok között (például `u*z`) mindig elvégezhető a rövidebb vektor ciklikus felhasználásával.

9. `A <- matrix(c(1,4,8,10,1,4,8,10,1),3,3); sum(diag(A%*%t(A)))`

10. A 19×19-es esetnél a mátrix létrehozható a `c(0,1)` kételemű vektor ciklikus felhasználásával: `matrix(c(0,1),19,19)`. A 20×20-as azonban nem, mert a 20 sort feltöltve (oszloponként) a vektor 10-szer felhasználódik, és a második oszlop ugyanúgy 0-val fog kezdődni, mint az első. Ezért itt ahhoz a trükkhöz folyamodunk, hogy először létrehozunk egy eggyel kevesebb (páratlan sok sorból álló) mártixot, és az utolsó sort hozzáragasztjuk: rbind(matrix(c(0,1),19,20),c(1,0)).

11. `u <- c(3,5,4,9); v <- c(6,3,8,2); u%*%v; sum(u %*% t(v))`

12. A trükk az, hogy létrehozunk egy vektort, ami eggyel hosszabb, mint a mátrix első oszlopa, és ezt használjuk fel ciklikusan, így az első elem minden oszlopban eggyel lejjebb kerül:

    `matrix(c(2,1,rep(0,n-2),1), n,n)`

13. `u <- rep((1:6), times = 6) v <- rep((1:6), each = 6) matrix(u^v, 6,6)`

6. fejezet

Adat- importálás, exportálás, working directory

Working directory:

A working directory az a könyvtár, melyet a program alapértelmezettnek tekint fájlok importálása és exportálása esetén. Ha ezen műveletek előtt ezt beállítjuk, nem szükséges a teljes elérési út beírása minden esetben. `getwd()` megadja a working directory-t, `setwd()`-vel át lehet állítani a working directory-t. Példa: `setwd("C:\\R")`. [1]

Adatok beolvasása:

A `read.table` parancs segítségével lehetőségünk van szöveges fájlokat (leginkább .txt, .csv) importálni.

- `read.table(file, header = FALSE, sep = "", quote = "\"'", dec = ".")`
  - alapértelmezett szeparátor a `TAB`
- `read.csv(file, header = TRUE, sep = ",", quote = "\"", dec = ".", fill = TRUE, comment.char = "", ...)`
  - alapértelmezett szeparátor a `,`
- `read.csv2(file, header = TRUE, sep = ";", quote = "\"", dec = ",", fill = TRUE, comment.char = "", ...)`
  - alapértelmezett szeparátor a `;`

[1] A working directory beállítása helyett célszerűbb projectet létrehozni.

RStudio-ban sokkal egyszerűbben beolvashatóak a táblák az Environment ⇒ Import Dataset segítségével.

Más adattáblák, fájltípusok beolvasási lépései itt olvashatóak részletesebben:
https://www.datacamp.com/community/tutorials/r-data-import-tutorial#gs.u9Wz5uQ

Adatok exportálása:

`write.table(x, file = "")` paranccsal exportálhatóak ki az általunk létrehozott táblák.

6.1. SQL lekérdezésekhez hasonló feladatok - Data.frame

Az R Data.frame objektumával a relációs adatbázis-kezelők minden mechanizmusa (szűrés, kiválasztás, összegzés, táblakapcsolás) megvalósítható. Az alábbi feladatok az SQL lekérdezésekkel ekvivalens eljárások használatára szolgálnak példaként.

6.1.1. Feladatok

1. FELADAT

   Olvassuk be az ugyfel.xlsx fájlt! A születési dátum mező adattípusa nem megfelelő. Alakítsuk át!

2. FELADAT

   Listázzuk ki az 1950 előtt született nők vezeték- és keresztnevét!

3. FELADAT

   Számoljuk ki az átlagéletkort nemek szerinti bontásban!

4. FELADAT

   Arra a kérdésre keressük a választ, hogy a női és férfi ügyfelek között látható-e statisztikai eltérés a megadott igazolvány típusban. Készítsünk egy kereszttáblát a nem és az igazolvány típusa szerint! Mivel a két adat különböző táblákban található, ezért a táblákat az ügyfelek egyedi azonosítója alapján össze is kell kapcsolni.

5. FELADAT

   Hány ügyfelünk van?

6. FELADAT

   Készíts egy új táblát, azoknak az ügyfeleknek a nevével (vezetéknév és keresztnév együtt), akik 1950. január 1. előtt születtek!

7. FELADAT

   Készíts egy új táblát, amely megadja minden ügyfélről, hogy hány éves lesz 10 év múlva!

8. FELADAT

   Adjuk hozzá a táblákhoz Pistikét, akinek adatai a következők:

vnev	knev	szdatum	nem	igazolvanytipus_id	igazolvanyszam
Kiss	Pisti	1990-01-01	M	SZ	123456AP

9. FELADAT

   Módosítsuk Pistike születési dátumát 1991. január 1-jére!

10. FELADAT

    Pistike sajnos megbukott a VPIR vizsgán. Töröljük ki az adatbázisból!

6.1.2. Megoldások

1. Az adatbeolvasás megoldható közvetlenül az Excelből való beolvasással, amihez használhatjuk például a `readxl` package-t:

   ```
   library(readxl)
   ugyfel <- read_excel("ugyfel.xlsx", col_types = c("numeric", "text", "text",
   "date", "text"))
   ugyfel2 <- read_excel("ugyfel.xlsx", sheet = "ugyfel2")
   ```

   Vagy lementhetjük az Excelből a két munkalap adatait valamilyen *plain text file* formátumban - például .csv - formátumban és beolvashatjuk így:

   ```
   ugyfel <- read.table("ugyfelcsv1.CSV", header = T, sep = ";")
   ugyfel2 <- read.table("ugyfelcsv2.CSV", header = T, sep = ";")
   ```

   A születési dátumot az `ugyfel$szdatum <- as.Date(ugyfel$szdatum)` utasítással lehet dátum formátumra konvertálni. A táblában lévő változók típusát például a `str(ugyfel)` utasítással ellenőrizhetjük, amit célszerű is megtenni, mielőtt tovább dolgoznánk vele.

2. ```
   F_50_elott <- ugyfel$szdatum <= as.Date("1950-01-01") & ugyfel$nem == "F"
   ugyfel[F_50_elott, c("vnev","knev")]
   ```

3. Dátumok kezelése az egyik legkellemetlenebb feladat, a következő példák az évben megadott életkor kiszámolására mind hasznosak lehetnek:
   ```

```
ugyfel$szev <- as.integer(substr(ugyfel$szdatum,1,4))
ugyfel$eletkor <- as.integer(format(Sys.Date(), "%Y")) - ugyfel$szev
```

vagy:

```
ugyfel$eletkor <- as.numeric(floor((Sys.Date()-ugyfel$szdatum)/365))
```

Ha ez megvan, a nemenkénti átlagot a következőképpen kaphatjuk meg beépített függvényekkel:

```
aggregate(formula=eletkor~nem, data = ugyfel, FUN = mean)
```

4. A táblák kapcsolására a

   ```
   tabla <- merge(ugyfel, ugyfel2, by.x = "ugyfel_id", by.y="ugyfel_id")
   ```

   utasítás szolgál. A kapcsoláshoz használt kulcsokat nem kötelező megadni, ez esetben az R a fejlécek neve alapján ”kitalálja”, tehát jelen esetben elég a következő kód is:

   ```
   tabla <- merge(ugyfel, ugyfel2)
   ```

   A kereszttábla különböző verziói így készíthetőek el:

   ```
   kimutatas <- table(tabla$igazolvanytipus_id,tabla$nem)
   prop.table(kimutatas)
   prop.table(kimutatas,1)
   prop.table(kimutatas,2)
   ```

5. `nrow(ugyfel)` vagy `length(ugyfel$ugyfel_id)`

6. `apply(ugyfel[ugyfel$szdatum <= as.Date("1950-01-01"),c("vnev","knev")],1,FUN = function(x) paste(x,collapse = " "))`

7. `cbind(ugyfel[,c("ugyfel_id","vnev","knev")],ugyfel$eletkor+10)`

8. ```
   ugyfelPisti <- rbind(ugyfel,data.frame(ugyfel_id = nrow(ugyfel)+1, vnev = "Kiss",
   knev="Pisti",szdatum=as.Date("1990-01-01"),nem = "M", szev = 1990 ,eletkor = 2017-1990 ))
   ugyfelPisti2 <- rbind(ugyfel2,data.frame(ugyfel_id = nrow(ugyfel)+1,
   igazolvanytipus_id="SZ" ,igazolvanyszam="123456AP" ))
   ```

9. Data.frame-ben az adatokat egyesével átírhatjuk a szokásos mátrixos indexeléssel. Ha azonban egyszerre több oszlop értékét akarjuk felülírni, akkor a beillesztendő adatokat egy kompatibilis (megfelelő típusú adatokat tartalmazó) Data.frame-nek kell tartalmaznia:

   ```
   ugyfelPisti[ugyfelPisti$ugyfel_id == 11,c(4,6,7)] <- data.frame(as.Date("1991-01-01")
   ```
   ```

10. Data.frame-ből változót (azaz oszlopot) direktben törölhetünk, ha az oszlopot egyenlővé tesszük a NULL változóval. Rekordokat (sort) azonban így nem lehet törölni. Ehhez le kell szűrni az eredeti táblát úgy, hogy a Pistikét ne tartalmazza:

```
ugyfelPistiTorolve <- ugyfelPisti[ugyfelPisti$ugyfel_id != 11,]
```

```
ugyfelPistiTorolve2 <- ugyfelPisti2[ugyfelPisti2$ugyfel_id != 11,]
```

6.2. SQL lekérdezésekhez hasonló feladatok - Data.table

Ugyanezen feladatok Data.table-el való megoldása:

1. FELADAT

```
library(readxl)
```

```
ugyfel <- read_excel("ugyfel.xlsx", col_types = c("numeric", "text", "text",
"date", "text"))
```

```
ugyfel2 <- read_excel("ugyfel.xlsx", sheet = "ugyfel2")
```

```
ugyfel$szdatum <- as.Date(ugyfel$szdatum)
```

```
str(ugyfel)
```

```
library(data.table)
```

```
ugyfelDT <- setDT(ugyfel)
```

```
ugyfel2DT <- setDT(ugyfel2)
```

2. FELADAT

két oszlop kiválasztása

```
szuresDT <- ugyfelDT[,.(vnev,knev)]
```

csak nők kiválasztása

```
szuresDT <- ugyfelDT[nem == "F"]
```

megoldás

```
szuresDT <- ugyfelDT[nem == "F" & szdatum <=as.Date("1950-01-01"),.(vnev,knev)]
```

3. FELADAT

```
ugyfelDT[,eletkor:= as.numeric(floor((as.Date("2018-10-10")-szdatum)/365))]
```

```
ugyfelDT[,mean(eletkor),by=nem]
```

4. FELADAT

   ```
   tablaDT <- merge(ugyfelDT, ugyfel2DT)
   ```

   Hány férfi és női ügyfél van?

   ```
   kimutatasDT1 <- tablaDT[,.N,by=nem]
   ```

   .N -speciális utasítás a sorok számára

   ```
   kimutatasDT2<-tablaDT[,.N,by=.(nem,igazolvanytipus_id)]
   kimutatasDT2[,Narany:= N/nrow(ugyfelDT)]
   ```

5. FELADAT

   ```
   ugyfelDT[,.N]
   ```

6. FELADAT

   ```
   valtozoegyutt <- ugyfelDT[szdatum <= as.Date("1950-01-01"),.(paste(vnev, knev))]
   ```

7. FELADAT

   ```
   ugyfel10DT <- ugyfelDT
   ugyfel10DT <- ugyfel10DT[,ugyfeleletkorplusz10:=eletkor+10]
   ```

8. FELADAT

   ```
   ugyfelPistiDT <- rbind(ugyfelDT,list(11, "Kiss","Pisti",as.Date("1990-01-01"),"M",
   28 ,38))
   ugyfelPisti2DT <- rbind(ugyfel2DT,list(11,"SZ" ,"123456AP" ))
   ```

9. FELADAT

   ```
   ugyfelPistiDT[ugyfel_id==11,szdatum:=as.Date("1991-01-01")]
   ugyfelPistiDT[ugyfel_id==11,eletkor:=27]
   ```

10. FELADAT

    ```
    ugyfelPistiTorolveDT <- ugyfelPistiDT[ugyfel_id!=11]
    ugyfelPistiTorolve2DT <- ugyfelPisti2DT[ugyfel_id!= 11]
    ```

7. fejezet

Grafikus alkalmazások alapjai

Az emberi érzékszervek működési mechanizmusainak következtében agyunk legkönnyebben a vizuálisan megjelenített információt képes feldolgozni. Az R grafikus interfésze kifejezetten magasan fejlett és felhasználóbarát, szinte kimeríthetetlen lehetőségeket biztosít az adatok kettő, három, vagy akár négy dimenzióban való megjelenítésére. Ebben a fejezetben megismerkedünk ennek legelemibb eszközeivel.

7.1. Grafikus megjelenítés alapfüggvényei

A legfontosabb függvényeket az alapértelmezetten telepítésre kerülő és minden indításnál automatikusan betöltődő `graphics` package tartalmazza. Ezek egy része alacsony szintű funkciót lát el (például egy pontot rajzol, vagy egy vonalat húz egy már meglévő ábrára), másik részük olyan összetett feladatokat valósít meg, mint egy gráf vagy egy döntési fa megjelenítése. Egy rövid lista azokról a függvényekről, melyeket érdemes fejből is ismerni:

- `windows` vagy `plot.new` : Új, üres ablakot nyit, ahova rajzolni lehet. A megnyitott ablakok száma korlátozva van, a jelenlegi verzióban 60 darabra.
- `frame` : Letörli az aktuális rajzot az aktív ablakból anélkül, hogy bezárná az ablakot, így új ábrát rajzolhatunk rá. Főként akkor hasznos, ha animációt akarunk készíteni, melynek minden filmkockáját újra kell rajzolni.
- `plot` : Generikus függvény, különböző objektumok megjelenítésére, például két (azonos hosszúságú) adatsort ábrázol egymás függvényében, megrajzolja egy függvény grafikonját, de akár egy neurális hálót is megjelenít, ha olyan objektumot adunk át neki.
- `matplot` : Több adatsort ábrázol egyazon koordináta-rendszerben. Az adatokat egy mátrix oszlopaiként kell átadni.

- `curve` : Egy kifejezést (melyben x változó szerepelhet, mint formális paraméter) ábrázol vonaldiagramként. Állhat önmagában, de már létező ábrára is rárajzolhatunk vele egy új görbét.
- `points` : Már létező ábrára rakhatunk vele pontokat a megfelelő koordinátákra.
- `abline` : Az $y = ax + b$ egyenletű egyenes grafikonját teszi a már meglévő ábrára, illetve a h vagy v paraméterek megadásával vízszintes, illetve függőleges egyenes is rajzolható.
- `lines` : Töröttvonalat rajzol a megadott pontok összekötésével a már meglévő ábrára.
- `polygon` : Sokszöget rajzol a már meglévő ábrára. Lényegében ugyanaz, mint a `lines`, csak az utolsó pontot köti össze az elsővel, és ki tudja színezni a sokszög belsejét is.
- `segments` : Hasonló, mint a `lines`, csak a szakaszok kezdő és végpontja is megadható különkülön (a `lines` esetén a következő szakasz kezdőpontja megegyezik az előző végpontjával, ennél viszont külön kell őket megadni).
- `arrows` : Hasonló, mint a `segments`, csak nyilakat rajzol szakaszok helyett.
- `barplot` : Oszlop- vagy sávdiagramot készít különböző beállításokkal. Külön, üres ablakba teszi az ábrát.
- `hist` : Hisztogramot készít és alapértelmezetten automatikusan ábrázolja is.
- `boxplot` : Értelemszerűen boxplotot rajzol.
- legend : Felhasználóbarát módon testreszabható jelmagyarázatot ad az ábrához.

A fenti függvényekre mindenképpen érdemes használat előtt a help-ben rákeresni, mert ezerféleképpen paraméterezhetőek és nem könnyű (és nem is érdemes) fejben tartani minden lehetséges beállítást (megadható a rajzolt pontok, vonalak típusa, színe stb.). Nem minden grafikus paraméter adható át közvetlenül a *high-level* függvényeknek, mint például a `plot` vagy `matplot`. Ezeket közvetlenül az új ablak megnyitása után a `par` utasítással lehet kiolvasni vagy átírni.

7.2. Grafikus megjelenítéshez kapcsolódó feladatok

Az alábbi feladatok a 7.1 fejezetben felsorolt függvények segítségével könnyen megoldhatóak. A fejezet végén egy-egy lehetséges megoldást is feltüntettünk.

7.2.1. Feladatok

1. FELADAT

   Definiáljunk egy 10 000 hosszúságú vektort, a (-3pi,3pi) intervallumon és ennek segítségével ábrázoljuk a sin és cos függvény grafikonját! Az ábrára írjuk rá a függvények nevét is, mint címet!

2. FELADAT

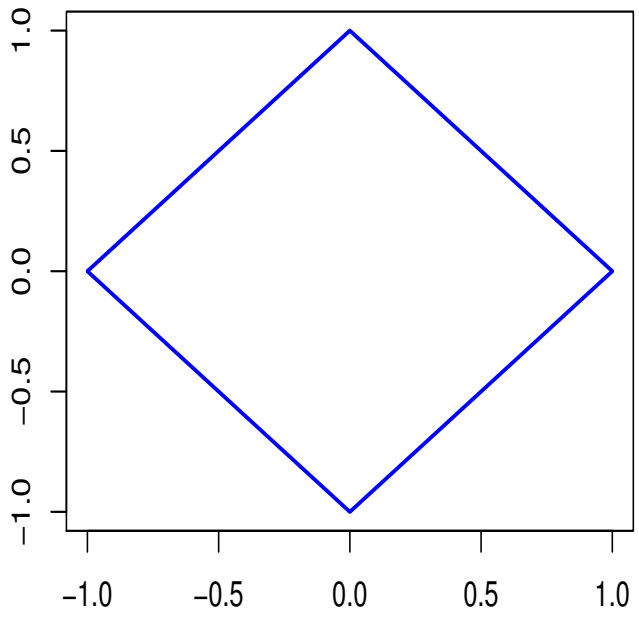
   Ábrázoljuk közös koordináta-rendszerben a sin(x) és cos(x) függvényt egyszerre, különböző színnel!

3. FELADAT

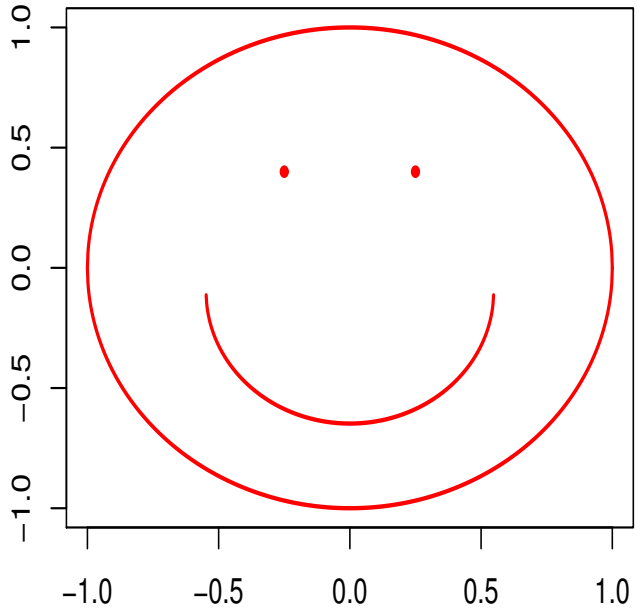
   Definiáljunk egy 10 000 hosszúságú vektort, a (-3pi,3pi) intervallumon és ennek segítségével ábrázoljuk a sin függvény pozitív részét (tehát azt a függvényt, ami sin(x)–szel egyenlő, ha ez pozitív és nullával egyenlő, amikor sin(x) negatívba megy)!

4. FELADAT

   Írjunk kódot, ami elkészíti a következő ábrát:



5. FELADAT Írjunk kódot, ami elkészíti a következő ábrát:



7.2.2. Megoldások

1. Feladat

```
x <- seq(-3*pi,3*pi,length.out = 10000)
windows(20,8); par(mfrow=1:2);
plot(x,sin(x), main = "szinusz", type = "l");
plot(x,cos(x), main = "koszinusz", type = "l")
```

2. Feladat

```
x <- seq(-3*pi,3*pi,length.out = 10000)
windows(10,6);matplot(x,cbind(sin(x),cos(x)), type = "l", lty = 1)
```

3. Feladat

```
x <- seq(-3*pi,3*pi,length.out = 10000)
y <- sin(x);
```

```
y[y < 0] <- 0;
windows(10,6);
plot(x,y)
```

vagy egyszerűbben:

```
windows(10,6)
plot(x,pmax(y,0))
```

4. Feladat

```
x <- seq(-1,1,length = 1000)
matplot(x,cbind(1-abs(x), abs(x)-1), typ = "l", lwd = 2, col = 4, lty = 1, xlab
= "", ylab = "")
```

vagy

```
windows()
plot(NA, xlim=c(-1,1), ylim=c(-1,1))
polygon (c(-1,0,1,0), c(0,-1,0,1))
```

5. Feladat

```
x <- seq(-1,1,length = 1000)
matplot(x,cbind(sqrt(1-x^2),-sqrt(1-x^2),-sqrt(0.3-x^2)-0.1), typ = "l", lwd
= 2, col = 2, lty = 1, xlab = "", ylab = "")
points(c(-0.25,0.25),c(0.4,0.4),pch = 20, col = 2)
```

8. fejezet

Algoritmusok és vezérlési szerkezetek

Algoritmus alatt – hogy ne vesszünk el a matematikai absztrakció útvesztőiben – számítógéppel mechanikusan végrehajtható utasítássorozatot értünk. Hogy a kicsit sem precíz definícióval is dolgozni tudjunk, meg kell határoznunk, hogy a számítógép milyen utasítások végrehajtására képes. Ehhez jó szemlélet a következő. Adott egy véges sok elemből álló lista, elemei megszámozva 1-től n-ig. A lista mindegyik eleme a következő utasítások valamelyikét tartalmazhatja:

- Valamilyen egyszerű aritmetikai művelet, azaz két vagy több számmal négy alapműveletből (esetleg gyökvonásból) felépíthető algebrai kifejezés kiszámítása és eltárolása egy változóba. A kifejezés tartalmazhatja korábban kiszámolt részfeladatok végeredményét is, melyet egy korábbi lépésben létrehozott változóban tároltunk.

- Egy egyértelműen eldönthető $a < b$ vagy $a \leq b$ alakú matematikai reláció kiértékelése (annak eldöntése, hogy igaz vagy hamis az állítás), ahol a és b egy-egy korábban kiszámolt változó (vagy már előre adott input adat), és ugrás a lista egy adott sorszámú utasítására, amennyiben az állítás igaz.

- STOP vagy EXIT utasítás és opcionálisan egy visszatérési érték, output, aminek a kiszámolására az algoritmusa létrejött.

A számítógép az algoritmus futtatása során mindig a lista elején kezdi a program végrehajtását, és akkor áll le, ha kilépési pontra ér, vagy a lista utolsó elemére, ha ez nem tartalmaz olyan utasítást, hogy egy korábbi pontra kell visszaugrani. Az algoritmus alapértelmezetten mindig a következő utasításra ugrik, ha máshová való ugrásra nincs utasítás az adott pontban.

A számítógép tehát képes aritmetikai műveleteket és összehasonlítást végezni, vagyis ki tudja számolni, hogy $2 + 2 = 4$, és az $5 \leq 7$ kérdésre képes megválaszolni, hogy `TRUE`. Általában azt a problémát tekintjük algoritmikusan megoldhatónak, amelyre fel tudunk írni egy ilyen egyszerű kérdésekből és utasításokból álló véges hosszú sorozatot.

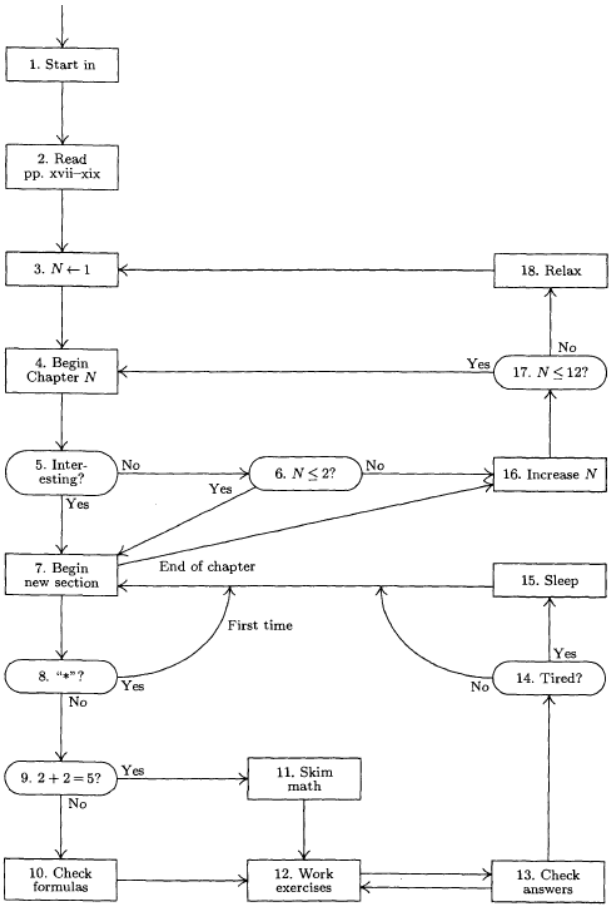
A következő algoritmus Donald Knut Art of Computer Programming című könyvének bevezetőjében található, arról, hogyan kell a könyvet elolvasni:

1. Begin reading this procedure, unless you have already begun to read it. *Continue to follow the steps faithfully.* (The general form of this procedure and its accompanying flow chart will be used throughout this book.)
2. Read the Notes on the Exercises, on pages xv–xvii.
3. Set N equal to 1.
4. Begin reading Chapter N. Do *not* read the quotations that appear at the beginning of the chapter.
5. Is the subject of the chapter interesting to you? If so, go to step 7; if not, go to step 6.
6. Is $N \leq 2$? If not, go to step 16; if so, scan through the chapter anyway. (Chapters 1 and 2 contain important introductory material and also a review of basic programming techniques. You should at least skim over the sections on notation and about `MIX`.)
7. Begin reading the next section of the chapter; if you have already reached the end of the chapter, however, go to step 16.
8. Is section number marked with "*"? If so, you may omit this section on first reading (it covers a rather specialized topic that is interesting but not essential); go back to step 7.
9. Are you mathematically inclined? If math is all Greek to you, go to step 11; otherwise proceed to step 10.
10. Check the mathematical derivations made in this section (and report errors to the author). Go to step 12.
11. If the current section is full of mathematical computations, you had better omit reading the derivations. However, you should become familiar with the basic results of the section; they are usually stated near the beginning, or in *slanted type* right at the very end of the hard parts.
12. Work the recommended exercises in this section in accordance with the hints given in the Notes on the Exercises (which you read in step 2).
13. After you have worked on the exercises to your satisfaction, check your answers with the answer printed in the corresponding answer section at the rear of the book (if any answer appears for that problem). Also read the answers to the exercises you did not have time to work. *Note:* In most cases it is reasonable to read the answer to exercise n before working on exercise $n + 1$, so steps 12–13 are usually done simultaneously.
14. Are you tired? If not, go back to step 7.
15. Go to sleep. Then, wake up, and go back to step 7.
16. Increase N by one. If $N = 3, 5, 7, 9, 11$, or 12, begin the next volume of this set of books.
17. If N is less than or equal to 12, go back to step 4.
18. Congratulations. Now try to get your friends to purchase a copy of Volume 1 and to start reading it. Also, go back to step 3.

Bármilyen primitívnek is tűnik a fenti ”definíció”, a számítógépek ennél (sokkal) többet nem tudnak. A manapság sokat hangoztatott *machine learning, mesterséges intelligencia, gépi tanulás* és hasonló ”fancy” kifejezések nem a számítógépek képességeinek, hanem az őket programozó alkotó emberi szellemnek a megtestesülései.

Emellett érdemes itt megjegyezni, hogy noha az első számítógépek megalkotása óta a számítási kapacitás a többszörösére (több ezerszeresére) nőtt, a számítógépek alapvető képességei nem változtak. Magyarul: a számítógép nem tud többet, mint amennyit a hatvanas években tudott, csak az egységnyi helyre zsúfolható tranzisztorok számának rohamos növekedésével gyorsabban és több adattal képes ugyanazt kiszámolni. Ez természetesen csak a számítógép fizikai tulajdonságaira vonatkozik, algoritmusok létezhetnek és léteznek is olyanok, amik ötven (vagy öt) évvel ezelőtt nem voltak, tehát a szoftverek nyilván folyamatosan fejlődnek.

Az emberi szem/agy számára könnyebben értelmezhető egy algoritmus, ha ún. flow diagrammal szemléltetjük. Az Art of Computer Programming-ban lévő algoritmus grafikusan megjelenítve így fest:



Flow chart for reading this set of books.

Egy algoritmus tehát inputadatokból, véges sok elemi utasításból és azok (bizonyos feltételektől függő) végrehajtási sorrendjére vonatkozó szabályokból áll.

Egy algoritmus egy általános lépése tehát vagy egy elemi utasítás, vagy egy feltétel teljesülésétől függően ugrás az algoritmus egy másik pontjára. Az ilyen lépésekből álló utasítássorozatok le is programozhatók, ugyanis a számítógép processzora is ilyen utasítások végrehajtására képes.

A programozás (h)őskorában ezt valóban így is csinálták, azonban a modern programozási nyelvek jelentős része nem tartalmazza ezt az ún. GoTo típusú utasítást, mely máig parázs viták tárgyát képezi, és egyes vélekedések szerint túl veszélyes az átlag programozónak.

Egy ilyen algoritmus írásakor valóban kiemelt figyelmet kellett fordítani arra, hogy a részfeladatok megoldásánál ne írjunk olyan memóriaterületre, amelyen még fontos adat található, az algoritmus minden pontján rendelkezésre álljon az az adat, amire éppen szükségünk van, illetve minden lehetséges kimenetel esetén egyértelműen megadjuk, mi a teendője a programnak. Ennek mélyebb tárgyalásába nem bocsátkozunk, az érdeklődőknek javasoljuk Edgar Dijkstra: Go To Statement Considered Harmful

című írását.

8.1. Algoritmusok építőkövei

A programok jobban áttekinthetővé és könnyebben értelmezhetővé válnak, ha nem engedjük meg tetszőleges flow chart használatát egy probléma leírására, hanem csak speciális szerkezetű gráfokat használunk föl. Letiltjuk tehát az algoritmusok írása során az olyan utasítás használatát, mely a program utasításait tartalmazó lista adott sorszámú elemére való ugrást eredményezi (GoTo). Ennek az össze-vissza ugrálgatásnak a lehetősége a program egy random pontjára túl sok hiba forrása. Mi az az utasításkészlet, amit feltétlenül meg kell tartanunk ahhoz, hogy legyen esélyünk bármit is leprogramozni? Próbáljuk meg sora venni:

Szekvencia A számítógép a program utasításait tartalmazó lista elemein alapértelmezetten elejétől a végéig sorban megy végig. Tehát egyszerű utasítások (lineáris) sorozatát egy algoritmus nyilván mindenképpen tartalmazza.

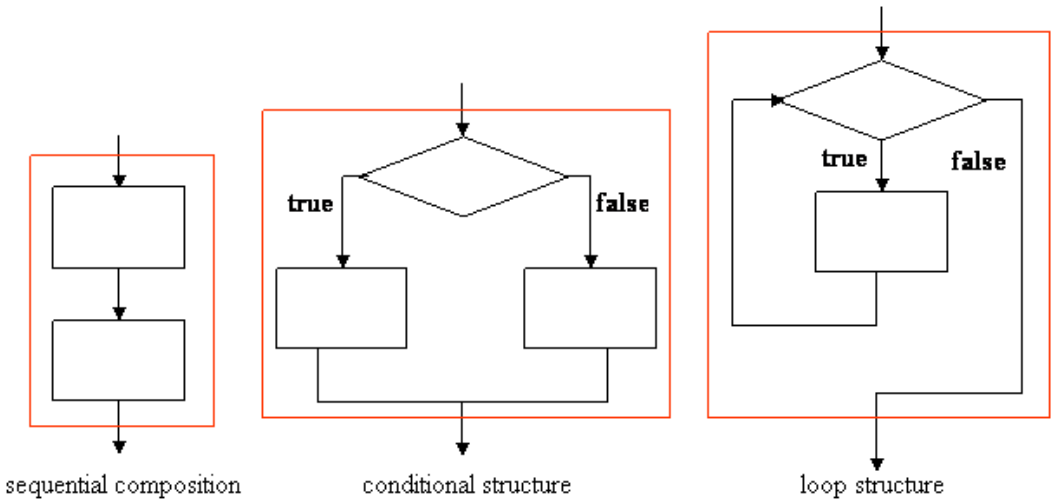
Feltételes elágazás (selection) Ahhoz, hogy egy program ne minden inputra pontosan ugyanazt az eredményt hozza ki, nyilván szükséges, hogy az algoritmus futása során egyes pontokon valamely feltételtől függően dönthessen, hogy két adott utasítás közül az egyiket vagy a másikat akarja-e végrehajtani. Ez azt jelenti, hogy egy feltételtől függően a program kétfelé ágazik, a két ág közül pontosan az egyik hajtódik végre, majd a két ág ismét egyesül, és ezen a ponton folytatódik a program futása.

Iteráció A szekvencia és az elágazás még nem teszi lehetővé olyan program írását, mely több utasítást hajt végre, mint amilyen hosszú. Egy algoritmus tervezése során gyakori eset, hogy ugyanazt az egy utasítást kell sokszor, közvetlenül egymás után megismételni, egészen addig, amíg valamely feltétel nem teljesül. Ezt nevezzük iterációnak: egy adott utasításokból álló blokk végrehajtása mindaddig, amíg egy változó értéke igaz. Ez tehát két nagyon speciális GoTo utasításnak felel meg. Az első először ellenőrzi egy feltétel teljesülését, és ha az nem teljesül, akkor előreugrik a programban, azaz kihagy egy néhány utasításból álló blokkot. Ha a feltétel teljesül, akkor ez az utasításokból álló blokk végrehajtódik és ennek az utolsó utasítása, hogy ugorjon vissza a blokk elé a feltétel ellenőrzésének helyére (és ismételje meg az egészet).

A modern programozási nyelvek nagy része a GoTo utasítás helyett ennek a három speciális szerkezetnek a használatát teszi lehetővé. Ennek létjogosultságát az alábbi matematikai eredmény adja:

Tétel (Structured program theorem - Böhm-Jacopini, 1966)**.** *Minden algoritmus megvalósítható olyan flow chart-tal is, mely csupán a fenti három ún. vezérlési szerkezetet tartalmazza.*

A tétel szerint bármilyen bonyolult diagram írja is le egy algoritmus megoldását, megadható egy azzal ekvivalens diagram, mely csupán ezt a három szerkezetet kombinálja:



A tételt úgy kell érteni, hogy az algoritmust leíró flow chart kizárólag ilyen szerkezeteket tartalmaz tetszőleges mélységben egymásba ágyazva.

8.2. Vezérlési szerkezetek R-ben

A legtöbb ma használt programozási nyelv a Böhm-Jacopini-tétel szerinti szerkezeteket megvalósító nyelvi elemeket és néhány plusz ”kényelmi funkciót” tartalmaz. A tétel szerint bármely két programozási nyelv, amely ezt a három elemet tartalmazza, ekvivalens abban az értelemben, hogy pontosan ugyanazokat a dolgokat tudják kiszámolni (vannak problémák, amiket számítógéppel egyáltalán nem lehet megoldani). Ez azt is jelenti, hogy az egyes programozási nyelvek közti különbség (kis túlzással) csupán esztétikai kérdés, a velük megoldható problémák köre pontosan ugyanaz. Az R-ben a ?Control utasítás beírásával érhetjünk el a vezérlési szerkezetek leírását.

8.2.1. A Böhm-Jacopini-tétel szerinti utasításkészlet

1. A szekvenciális programfuttatás magától értetődő, a forráskódban lévő utasítások alapértelmezetten egymás után, azaz föntről lefelé, ezen belül balról jobbra hajtódnak végre, ehhez semmilyen

vezérlési szerkezetre nincsen szükség. R-ben minden utasítás végére pontosvesszőt kell tenni, vagy új sort kezdeni, ezen kívül semmi speciális tudnivaló nincs.

2. A feltételes elágazás talán minden programozási nyelvben megtalálható, és általában hasonlóan működik.

   Szintaktikája R-ben:

   `if(cond) expr, illetve if(cond) cons.expr else alt.expr`

   Az utasítás eredménye, hogy a feltétel (cond) kiértékelődése után a két egymást kizáró utasítás közül az első hajtódik végre (vagyis a cons.expr kifejezés értékelődik ki), ha a feltétel igaz, és a második (opcionális, azaz elhagyható, alternatív feltétel, vagyis alt.expr), ha hamis. A feltételes elágazásnál nem kötelező a kapcsos zárójelek használata, erre csak akkor van szükség, ha valamelyik ágban több utasítást akarunk végrehajtani.

   ```
   x <- 4
   if (x < 5) print("kicsi") else print("nagy")
   ```

   Vigyázzunk, hogy logikai feltételként olyan kifejezés is értelmezhető, amiről ez elsőre nem triviális. A következő kód például hiba nélkül lefut:

   ```
   x <- "négy"
   if (x < 5) print("kicsi") else print("nagy")
   ```

   Ha a kiértékelendő feltétel logikai értékként nem értelmezhető, akkor a program hibaüzenettel leáll. Próbáljuk ki ezt is:

   ```
   x <- NA
   if (x < 5) print("kicsi") else print("nagy")
   ```

   Ha a kiértékelendő feltétel egynél hosszabb, akkor csak az első komponensét veszi figyelembe, és erről warning-ot dob, de nem fut hibára.

   ```
   x <- 1:6
   if (x < 5) print("kicsi") else print("nagy")
   ```

3. Az iterációt megvalósító vezérlési szerkezet szintaktikája:

   `while(cond) expr`

   Ez mindaddig folytatja az expr kifejezés kiértékelését, amíg a cond feltétel igaz. A használata csak akkor szükséges, ha a kifejezés összetett. Nézzünk egy példát :

   ```
   a <- 137
   ```

   ```
   while(a != 1) if(a%%2 == 0) print(a <- a/2) else print(a <- 3*a+1)
   ```

   Induljunk ki egy tetszőleges természetes számból, majd ismételjük a következőket: ha a szám páros, elosztjuk kettővel, ha páratlan, megszorozzuk hárommal és hozzáadunk egyet; ez lesz az új szám. (A dupla %% operátor az első szám második szerinti osztási maradékát számolja ki.) Majd ismét, ha az új szám páros, osztjuk kettővel, ha páratlan, szorozzuk hárommal és hozzáadunk egyet. Ezt folytatjuk. Akkor állunk le, ha elérjük az 1-et. Az iterációk számát nem tudjuk előre megmondani, csak az aktuális számról tudjuk ellenőrizni, hogy egyenlő-e 1-gyel. A print utasítás csak azért kell, hogy lássuk a sorozat tagjait.

Noha a tétel szerint a fenti három utasítás használatával bármit le lehet programozni (mármint, amit egyáltalán le lehet), gyakran adódik olyan helyzet, amihez egy speciális (egyszerűbb) utasítás biztonságosabb, kevesebb hibalehetőséget rejt, vagy csak kényelmesebb, esetleg könnyebben értelmezhető, ha később rá kell jönni a kód logikájára. Az R-ben is van néhány ilyen utasítás.

8.2.2. Speciális vezérlési szerkezetek

1. Az iteráció legfontosabb és leggyakoribb speciális esete, amikor a ciklusmag egy előre ismert, véges halmaz elemeivel paraméterezhető (természetesen a ciklusmagnak nem szükséges valóban függnie ettől a paramétertől). Erre a speciális esetre R-ben külön utasítás van, melynek szintaktikája:

   `for (var in SEQ){ <utasítások> }`

   A *for* és *in* kulcsszavakat az RStudio kékkel kiemeli, az első ovális zárójel kötelező elem, a blokkot megadó kapcsos { } zárójel opcionális, csak akkor van rá szükség, ha a blokk több utasításból áll. A *var* tetszőleges formális paraméter, amely lehet korábban másra használt változónév, de lehet olyan is, melyet korábban nem definiáltunk. A *SEQ* egy kifejezés, amelynek értéke egy vektor, mátrix vagy lista (data.frame-et is beleértve), vagy bármely bonyolultabb objektum, melynek elemein végig lehet menni (lehet egyetlen elemű, de akár üres is). Ha ez egy változónév, akkor a változót előtte létre kell hozni, de lehet egy függvényhívás is, mely egy megfelelő objektumot ad vissza. Ilyen egyszerű iteráció például, hogy írjuk ki a számokat egy adott intervallumból:

   ```
   for (i in 3:7) print(i)
   ```

Fontos megjegyezni, hogy a `for` ciklus nem csupán egy felokosított `while` ciklus, hanem nagyon speciális. Az utasítás hatására a blokkban lévő utasítások annyiszor futnak le, ahány elemű a `SEQ` kifejezés értéke és a `var` változó minden egyes iterációban egyenlő a `SEQ` annyiadik elemével, ahányadszorra fut az iteráció. A ciklus előtt a `SEQ` kifejezés kiértékelődik, és a ciklusmagban történő esetleges felülírása nem befolyásolja az iterációt. Az `i` ciklusváltozó a ciklusmagban felülírható, azonban ez nem befolyásolja értékét a következő iterációban, mely annak kezdetekor mindenképpen megegyezik a `seq` soron következő elemével.

Például a következő utasítás nem okoz végtelen ciklust, csak kiírja a számokat 0-tól 9-ig (az *i* változó felülírása csak az aktuális iteráció erejéig hat:

```
for (i in 1:10) print(i <- i-1)
```

A következő kód sem fut hibára:

```
v <- 1:10
for (i in v) {print(i); v <- NULL}
```

A `v` változó felülírása az első iterációban nem hat a ciklusra.

2. A `break` utasítás az őt közvetlenül tartalmazó legbelső iterációs ciklusból való kilépést eredményezi. Ez a VBA *Exit For*, illetve *Exit Do* és hasonló utasítások megfelelője. Egymásba ágyazott iterációk esetén csak a legbelsőből lép ki. Leggyakrabban egy `if` feltétel belsejében alkalmazzuk, ekkor megszakítja a feltételt tartalmazó ciklust. A `for` ciklus használatához általában előre tudni kell, hogy a ciklusmag *pontosan* hányszor fog lefutni. A `for` és a `break` együttes használatával írhatunk olyan ciklust, amiről előre tudjuk, hogy *legfeljebb* hányszor fog lefutni (ez továbbra sem teljes értékű helyettesítője a `while` ciklusnak, de nagyon hasznos. Például, ha a 3. pontban lévő példánál elő akarunk írni egy fix határt, hogy legfeljebb hányszor fusson le a ciklus (megakadályozandó, hogy végtelen ciklusba kerüljön), akkor már `for` ciklussal is megírhatjuk a feladatot:

```
a <- 137
maxIt <- 25
for (i in 1:maxIt) {
     if(a%%2==0) print(a<- a/2) else print(a <- 3*a+1)
     if (a == 1) break
}
```

3. A `next` utasítás a C++ `continue` utasításának felel meg, és semmi köze a VBA-ból ismert Next kulcsszóhoz. Ennek hatására az aktuális iterációs ciklus hátralévő utasításai nem futnak

le, és a program azonnal a következő iterációs ciklusba lép. Az alábbi kód ezt használva kiírja a számokat egytől százig, úgy, hogy kihagy minden 7-el és 9-el oszthatót:

```
for (i in 1:100){if (i%%7 == 0 | i%%9 == 0) next; print(i)}
```

4. A `repeat` utasítás egy formálisan végtelen ciklus, mely kitartóan értékeli ki az utána következő kifejezést. `repeat expr` ugyanaz az utasítás, mint `while(TRUE) expr`, tehát nem túl hasznos, és `break` utasítás nélkül nem is használható, mert magától soha nem lép ki.

5. Van még két extra vezérlési szerkezet, melyekre a help-ben a See Also résznél található utalás. Ezek nem tekinthetőek szűkebb értelemben vezérlési szerkezetnek, inkább függvénynek. Az `ifelse` az `if` vektorizált változata, a switch függvény pedig leginkább a c++ `switch` vagy a VBA `Select Case` utasításának megfelelője. Egyik sem túl hasznos konstrukció.

8.2.3. Iteráció kiváltása vektorműveletekkel

Noha a fent tárgyalt vezérlési szerkezetek alkotják minden algoritmus építőköveit, használatuk R-ben lehetőség szerint kerülendő és nem hatékony. Az R nyelv elemi változói a vektorok és a vektorokkal végzett műveletek komponensenként értelmezettek, így eleve iterációként működnek, csak ez a háttérben történik, ami általában gyorsabb, mint az általunk írt ciklusok. Vannak algoritmusok, melyek `while` ciklus nélkül nem programozhatóak le, és a rekurzív iterációk (ahol az i-edik lépés függ(het) attól, hogy mi történt az $(i-1)$-edikben vagy korábban) esetén sem elkerülhető a `for` ciklus. Az adatelemzésben tipikusnak mondható egyszerű ciklusok azonban gyakran megvalósíthatóak vektorműveletekkel.

Példaként adjuk össze 1-től 100-ig a páratlan számokat. Ez szinte bármely programozási nyelven megoldható a következő algoritmussal:

```
ptlSzamok <- 0
for (i in 1:100){
     if (i%% 2 == 1){
          ptlSzamok <- ptlSzamok + i
     }
}
```

R-ben azonban mind az iteráció, mind a páratlan számok kiválasztása "szimulálható" vektorműveletekkel:

```
sum((1:100)[1:100 %% 2 == 1])
```

8.3. Függvények

A függvény vagy szubrutin logikailag önálló egységet képező, illetve gyakran ismétlődő részfeladatok (mellékszámítások) elkülönítésére létrehozott kódrészlet. Ez a definíció szándékoltan és lényegesen

különbözik a függvény matematikai definíciójától, ugyanis egy függvény nem feltétlenül csak input adatokhoz rendel hozzá output adatokat, az elvégzett érdemi feladatok gyakran *side effect*-ként jelentkeznek. Annak oka, hogy a függvényeket ebben a fejezetben tárgyaljuk az, hogy a függvényhívás (leegyszerűsítve) általában egy GoTo utasítással jön létre úgy, hogy átmásolásra kerülnek a memóriában a függvénynek átadott változók, majd a vezérlés átugrik a függvény külön létrehozott kódjára, az utasítások lefutnak, végül a függvény által visszaadott output visszakerül egy megfelelő változóba, a program futása pedig visszaugrik a függvényhívás helyére és onnan folytatódik. Ennek következménye, hogy a függvényhívásnak minden programozási nyelvben költségei vannak, ami alatt azt értjük, hogy több művelet elvégzését igényli (s ebből kifolyólag tovább tart), mint ha ugyanazokat az utasításokat sorban beleírnánk a kódba a függvényhívás helyére. Ez a hatás az R-ben különösen durva, a függvények hívása nagyon sok "háztartási információ" generálásával jár, amely az R rendkívüli rugalmasságának ára. Ennek ellenére függvényeket írni jól definiált részproblémák megoldására több előnnyel jár, mint hátránnyal, mert gyakran ismétlődő feladatok megoldását elegendő egyszer jól megírni, és onnantól bármikor újrahasznosítható, illetve másokkal is megosztható (természetesen nem a ZH-n a többiekkel). Emellett jobban átlátható, strukturált kód írását teszi lehetővé, amely hosszú távon nagy valószínűséggel megéri.

8.3.1. Függvények R-ben

A függvény az R-ben ugyanolyan objektum, mint bármi más, és ugyanolyan értékadással is jön létre. Ennek szintaktikája:

```
<függvény neve> <- function(<formális paraméterek felsorolása>){ <utasítások> }
```

Egyetlen kifejezésből álló függvény esetén a { } elhagyható. A függvények definiálásakor használható a `return` utasítás, ami azonnal kilép a függvényből és visszatér az átadott kifejezés értékével. Ennek használata nem kötelező, a függvény visszatérési értéke mindig az utoljára kiértékelt kifejezés.

Mivel az R-ben a függvény ugyanolyan objektum, mint bármi más, függvényt létrehozhatunk függvényen belül is, sőt egy függvény visszatérési értéke is lehet egy másik függvény. Írhatunk például olyan függvényt, ami adott `x1, x2, y1, y2` paraméterekhez visszaadja azt a lineáris függvényt, amelynek grafikonja átmegy a `(x1; y1)` és `(x2; y2)` pontokon. A kapott függvénynek az alappontokat már nem kell minden függvényhívásnál átadni, csak a valódi változóját.

A függvény definícióját megadó kód csupán létrehozza a függvényt, mint objektumot, ha használni akarjuk, meg is kell hívni. Ez úgy történik, hogy beírjuk a függvény nevét és átadjuk neki a szükséges paramétereket. Nagyon gyakran előfordul, hogy olyan függvényeket használunk, melyeknek rengeteg (akár 10-15) paraméterük is van, ezért az R megengedi az opcionális paraméterek használatát, sőt ezt megkönnyítendő a függvényhívásnál lehetőség van (és ajánlott is) a paramétereknek név szerint értéket adni. Erre szolgál R-ben az "=" operátor. Ha ezt nem tesszük, az R az átadott paraméterek sorrendje

alapján próbálja meg „kitalálni”, hogy melyik melyik, így csak akkor lehetne egy paraméternek értéket adni, ha az összes megelőzőnek is adtunk.

Elrettentő példaként definiáljuk a következő függvényt:

```
f <- function(x = 0, y = 0) x+2*y
```

és próbáljuk ki az alábbi függvényhívásokat:

```
f(3);  f(x = 3);   f(y = 3);  f(y <- 3);  f(z = 3);  f(z <- 3)
```

Magyarázzuk meg, melyik függvényhívásnál, mi történik és miért, tovább miért lényeges a különbség az ”=” és a `"<-"` operátor között.

Az R megengedi a rekurzív függvényhívást, ami azt jelenti, hogy a függvény meghívhatja saját magát. A következő (nem túl értelmes) kód tehát elvileg tökéletes az $n!$ rekurzív kiszámolására:

```
nFact <- function(n){
     if (n <= 1) return(1)
     n*nFact(n-1)
}
```

Az R függvények látják a Global Environment-ben lévő változókat, tehát a következő kód tökéletesen hibátlan szintaktikailag, de feltehetően nem a szándékaink szerint fog működni:

```
R <- 0.2
f <- function(r) R^2
f(0.3)
```

8.4. Feladatok egyszerű algoritmusok írására

A következő feladatok mind nagyon egyszerűen megoldhatóak különböző iterációkkal és feltételes elágazással, de nagy részük vektorműveletekkel is. Törekedjünk arra, hogy a megoldás során ne írjunk felesleges ciklusokat, de azért könnyen áttekinthető, egyszerű kódot írjunk, amiről jövő hónapban is meg tudjuk mondani, mit csinál.

8.4.1. Feladatok

1. FELADAT

   Írjunk kódot, ami előállítja a következő 21×21-es mátrixot (a kódot úgy írjuk meg, hogy 21 helyett bármely páratlan számra működjön)!

```
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9 10
10  9  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  9 10
10  9  8  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  8  9 10
10  9  8  7  6  6  6  6  6  6  6  6  6  6  6  6  6  7  8  9 10
10  9  8  7  6  5  5  5  5  5  5  5  5  5  5  6  7  8  9 10
10  9  8  7  6  5  4  4  4  4  4  4  4  4  5  6  7  8  9 10
10  9  8  7  6  5  4  3  3  3  3  3  3  4  5  6  7  8  9 10
10  9  8  7  6  5  4  3  2  2  2  2  3  4  5  6  7  8  9 10
10  9  8  7  6  5  4  3  2  1  1  2  3  4  5  6  7  8  9 10
10  9  8  7  6  5  4  3  2  1  0  1  2  3  4  5  6  7  8  9 10
10  9  8  7  6  5  4  3  2  1  1  1  2  3  4  5  6  7  8  9 10
10  9  8  7  6  5  4  3  2  2  2  2  2  3  4  5  6  7  8  9 10
10  9  8  7  6  5  4  3  3  3  3  3  3  3  4  5  6  7  8  9 10
10  9  8  7  6  5  4  4  4  4  4  4  4  4  4  5  6  7  8  9 10
10  9  8  7  6  5  5  5  5  5  5  5  5  5  5  5  6  7  8  9 10
10  9  8  7  6  6  6  6  6  6  6  6  6  6  6  6  6  7  8  9 10
10  9  8  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  8  9 10
10  9  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  9 10
10  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
```

2. FELADAT

   Írjunk függvényt, ami egy megfelelő hosszú vektorban visszaadja a tízezernél nem nagyobb prímszámokat (használjuk az eratoszthenészi szitát)!

3. FELADAT

   A gonosz pasa tömlöcében 400 rab sínylődik. A pasa tömlöcének cellái úgy működnek, hogy ha egyet fordítanak a kulccsal a zárban, akkor kinyílik, ha előtte zárva volt, és bezáródik, ha előtte nyitva volt. Kezdetben a cellák zárva vannak. A pasa a születésnapján úgy dönt, hogy nagylelkű lesz, és szabadon engedi a rabokat, leküldi hát az egyik katonáját azzal a feladattal, hogy nyissa ki az összes cellát, azaz fordítson minden záron egyet a kulccsal. Később meggondolja magát, a kulcsok elforgatása után de még mielőtt még bármelyik rab elhagyhatná a celláját, leküld még egy katonát, azzal, hogy minden második cella ajtaján fordítson egyet a záron. Ismét meggondolja magát, és a következő katonát azzal az utasítással küldi le, hogy minden harmadik cella zárján fordítson egyet. Ez így megy tovább, egészen a 400-adik katonáig. Általában az i-edik katona azt az utasítást kapja, hogy minden i-edik cella (azaz minden i-vel osztható sorszámú cella) zárján egyet fordítson. Írjunk kódot, ami kiszámolja, hány rab szabadul ki és kik azok!

4. FELADAT

   Írjunk függvényt, ami vektorrá alakít egy szimmetrikus mátrixot, de csak a főátló alatti $\binom{n}{2}$ darab értéket tárolja oszlopfolytonosa!

   Tehát például a következőt számolja ki:

$$\begin{bmatrix} 1 & 5 & 6 & 7 \\ 5 & 2 & 8 & 9 \\ 6 & 8 & 3 & 0 \\ 7 & 9 & 0 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ 5 \\ 6 \\ 7 \\ 2 \\ 8 \\ 9 \\ 3 \\ 0 \\ 4 \end{bmatrix}$$

Írjuk meg ennek a függvénynek az inverzét, vagyis ami a vektorból visszaszámolja a szimmetrikus mátrixot!

8.4.2. Megoldások

1.
```
m <- matrix(0,21,21)
for (i in 1:21) for (j in 1:21) m[i,j] <- max(abs(i-11),abs(j-11))
```

2.
```
primek <- function(n){
     p <- rep(TRUE,n)
     p[1] <- FALSE
     for (i in 2:ceiling(sqrt(n))){
          if(p[i]) p[seq(2*i,n,by = i)] <- FALSE
     }
     (1:n)[p]
}
```

3.
```
n <- 400
cellak <- rep(FALSE,n)
for (i in 1:n) cellak[seq(i,400, by = i)] <- !cellak[seq(i,400, by = i)]
(1:n)[cellak]
```

4.
```
matrixbol_vektor <- function(m){
     v <- NULL
    for (i in 1:ncol(m)) v <- c(v, m[i:nrow(m),i] )
    v
}
vektorbol_matrix <- function(v){
    n=sqrt(2*length(v));
    A=matrix(0,n,n);
    for(i in 1:n){ A[i:n,i] <- v[1:(n-i+1)]; v <- v[-(1:(n-i+1))]}
    A
    B <- t(A)
    diag(B) <- 0
    A+B
}
```

9. fejezet

Véletlen számok

Mielőtt a technikai részletekben elmerülnénk, fontos megjegyezni, hogy bármilyen „értelmes” módon definiáljuk is a „véletlen” intuitív fogalmát, egyrészt, a tudomány jelenlegi állása szerint nem tudjuk, hogy létezik-e egyáltalán, másrészt, ha létezik is, a számítógép (mint determinisztikus szerkezet) akkor sem képes ilyesmit előállítani. Amikor „véletlenszám-generálásról” beszélünk, ezalatt olyan algoritmusokat értünk, melyek a véletlentől intuitíve elvárt, és a valószínűségszámítás eszközeivel bizonyított, statisztikai tulajdonságokat (pl. nagy számok törvénye, centrális határeloszlás-tétel) mutató/közelítő véges sorozatokat állítanak elő. A továbbiakban tehát „véletlen” számnak ezen algoritmusok eredményeként kapott álvéletlen sorozatokat nevezünk.

9.1. Véletlen számok R-ben

Véletlenszámok generálásához mindig négy függvény kapcsolódik: a sűrűségfüggvény, az eloszlásfüggvény, a kvantilisfüggvény (az eloszlásfüggvény inverze) és az adott eloszlásból véletlen mintát generáló függvény.

Írjuk be példaként a help-be: `?runif`. Itt láthatjuk ezt a négy függvényt az egyenletes (uniform) eloszlás esetére. Ugyanezt a négy függvényt láthatjuk exponenciális eloszlás (`rexp`), Student-féle t-eloszlás (`rt`), béta-eloszlás (`rbeta`), vagy Poisson-eloszlás (`rpois`) esetére és még számtalan egyéb eloszlásra. A függvények neve mindegyik eloszlás esetén `d` (density), `p` (probability), `q` (quantile) és `r` (random) betűkkel kezdődik, és az eloszlás nevének valamilyen rövidítésével végződik. Paraméterezésük függ az eloszlás típusától, és a help-ben benne van, hogy melyiket hogyan kell használni. Például egyenletes eloszlás esetén a `min`-t és `max`–ot kell megadni, normálisnál a szórást és a várható értéket, t-eloszlásnál a szabadságfokot, stb. A véletlenszámokat generáló függvény első argumentuma mindig a mintaelemek száma, ilyen hosszú vektort fog visszaadni. Az alábbi kód például tíz darab normális eloszlású véletlen számot generál 2 várható értékkel és 3 szórással:

```
rnorm(10,2,3)
```

Ha olyan eloszlásból akarunk mintát generálni, amit az R nem ismer, akkor természetesen működik az a trükk, hogy egyenletes eloszlást helyettesítünk az eloszlásfüggvény inverzébe (ha azt le tudjuk programozni), vagy előállítjuk ismert eloszlásokból, ha lehet. Ezzel kapcsolatban érdemes tudni, hogy a `runif` függvény nem generálja le a 0-t és az 1-et, tehát a nyílt intervallumból kapunk számokat, így nyugodtan behelyettesíthetjük olyan függvénybe, ami végtelent adna vissza a szélső pontokban.

A véletlenszám-generátor kezdőértéke beállítható a `set.seed` utasítással. Ez azt jelenti, hogy ugyanabból a kezdőpontból futtatja az algoritmust, ugyanazzal a paraméterezéssel, így ugyanazokat a számokat fogja újragenerálni minden futtatásnál (legalábbis ugyanazon R verzió használata esetén), így a "véletlenszámok" sorozata is reprodukálható. Ismételjük meg kétszer ugyanazt a véletlenszám-sorozatot:

```
set.seed(42); runif(10)
set.seed(42); runif(10)
```

9.2. Monte-Carlo szimuláció

Pénzügyi területen gyakori feladat a következő: adott egy Y valószínűségi változó, és egy $F : D \mapsto \mathbb{R}$ függvény, ahol D az Y értékkészlete. Határozzuk meg (becsüljük meg) az $X = F(Y)$ valószínűségi változó várható értékét, vagy valamely más tulajdonságát, akár az egész eloszlásfüggvényt!

Ebben a felírásban az Y valamilyen könnyen kezelhető kockázati faktor, az F függvény viszont nem lineáris, nem differenciálható, nem rendelkezik "szép" analitikus tulajdonságokkal ahhoz, hogy a transzformáció után kapott X változó (ami minket érdekel) analitikusan kezelhető legyen.

Erre példa, ha Y egy részvény hozamainak idősora egy adott időszakra, az X pedig egy, az időszak végén lejáró egzotikus opció (lehíváskori) értéke. Az Y változó könnyen kezelhető, általában feltételezzük, hogy független, normális eloszlású, az X eloszlása viszont nem ismert. Számoljuk ki az opció kifizetésének várható jelenértékét! (Ez bizonyos feltételek mellett az opció arbitrázsmentes ára.)

A probláma egy lehetséges közelítő megoldása, hogy "kellően nagy" elemszámú független mintát generálunk az X változóból, és a mintaátlaggal becsüljük a várható értéket. A módszer alkalmazhatóságának feltétele, hogy a változó mögötti kockázati faktor (Y) könnyen kezelhető, az F függvény pedig, ha matematikailag "ronda" is, konkrét értékeknél (géppel) könnyen kiszámolható legyen. Ekkor lehet X-ből mintát generálni. Konkrétan a példánál maradva az opció várható kifizetése megbecsülhető a következőképpen:

- Generáljuk le a részvényárfolyam egy lehetséges trajektóriáját (ehhez lényegében csak független normális eloszlású változókat (hozamokat) kell generálni!
- Számoljuk ki az opció kifizetését (ez könnyen megállapítható a trajektória ismeretében)!
- Ismételjük a fentieket "elég sokszor" és számoljuk ki a kapott számok átlagát!

9.3. Szimulációval megoldható feladatok

Az itt következő feladatok között már vannak trükkösek és nem triviálisak. A véletlen számok és a Monte-Carlo szimuláció a pénzügyekben gyakran alkalmazott technika, mind az ipari alkalmazások, mind az akadémiai kutatások terén, ezért érdemes gyakorolni.

9.3.1. Feladatok

1. FELADAT

   Generáljunk 10 000 darab véletlen számot 0-3 között egyenletes eloszlással! Becsüljük meg a sűrűségfüggvényét és eloszlásfüggvényét! Mit tapasztalunk a sűrűségfüggvénnyel kapcsolatban? Erre számítottunk?

2. FELADAT

   Generáljunk 10 000 darab véletlen számot, amely 70% eséllyel 1, 30% eséllyel 0! Ábrázold grafikusan az eredményt!

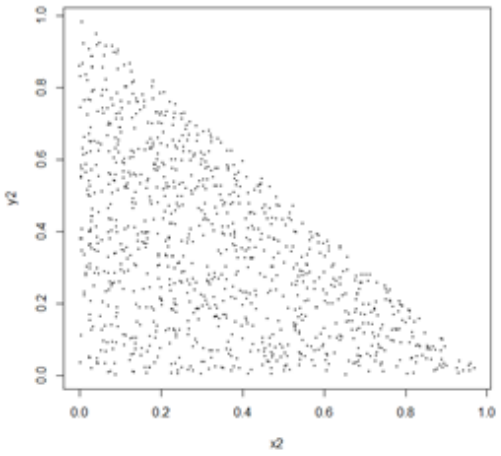
3. FELADAT

   Generáljunk 10 000 elemű normális eloszlású mintát, az átlag legyen 2, a szórás 0,3! Ábrázoljuk a sűrűségfüggvényét és eloszlásfüggvényét!

4. FELADAT

   Írjunk kódot, ami addig generál véletlen számokat egyenletes eloszlással `a [0,1]` intervallumból, amíg egy 0.99-nél nagyobb számot nem kap!

5. Generáljunk mintát abból a kétdimenziós eloszlásból, amely egyenletes az alábbi háromszögön!



   Mennyi a peremeloszlások korrelációs együtthatója?

6. FELADAT

   Írjunk függvényt, ami szimulációval becslést ad π értékére a következőképpen: generáljunk egyenletes eloszlásból megfelelően nagy mintát egy négyzet alakú tartományon, majd számoljuk össze azokat a pontokat, melyek a négyzetbe írt kör belsejébe esnek! Ezen pontok aránya a teljes mintában aszimptotikusan $\pi/4$. A szimulációt jelenítsük meg grafikusan is!

9.3.2. Megoldások

1. ```
   x <- runif(10000,0,3)
   windows(12,6); par(mfrow=1:2);
   plot(density(x)); plot(ecdf(x))
   ```

2. ```
   hist(as.numeric(runif(10000) > 0.3))
   ```

3. ```
   x <- rnorm(10000,2,0.3)
   windows(12,6)
   par(mfrow=1:2)
   plot(density(x))
   plot(ecdf(x))
   ```

4. ```
   a <- 0
   while(a < 0.99) print(a <- runif(1))
   ```

5. ```
   N <- 10000
   x <- runif(N)
   y <- runif(N)
   rossz <- y > 1 - x
   x2 <- c(x[!rossz],1 - y[rossz])
   y2 <- c(y[!rossz],1 - x[rossz])
   windows(); plot(x2, y2, pch = ".")
   print( cor(x2,y2) )
   ```

6. ```
   N <- 100000
   A <- matrix(runif(2*N,-1,1),N,2)
   b <- A[ , 1]^2+A[ , 2]^2 <= 1
   ```

```
windows();plot(A, pch = ".", col = b+1)
4*sum(b)/N
```

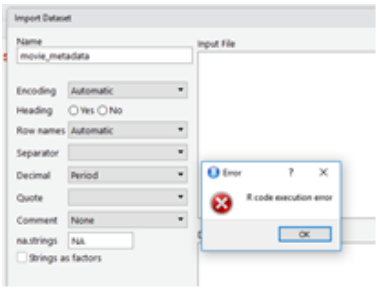
10. fejezet

Adatfeldolgozás, adattisztítás

10.1. Adatbeolvasási nehézségek

Fájlok beolvasása során számos nehézségbe ütközhetünk

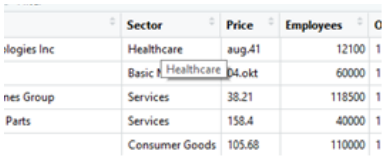
- a dokumentum elérési útjában tiltott karakterek találhatóak pl: í, Ã, ó, ő stb.



  A hibaüzenetben is látható hiba oka: `'C:/Users/Lilla/Desktop/tanÃt/movie_metadata.csv'` miközben a helyes mappa elérési út `C:\Users\Lilla\Desktop\tanít`.

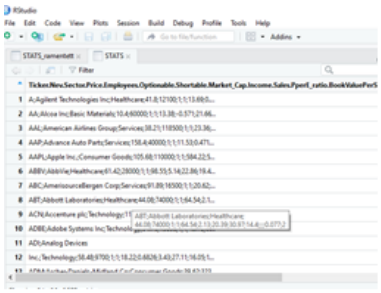
- Excelben rámentettünk a fájlra, és átalakultak a számok dátummá.

  Ez jellemzően akkor fordul elő, amikor ';' a szeparátor a decimális elválasztó pedig '.' (példa: `STATS_ramentett.csv`)



- szeparátor, decimális jelölő

  A fájl beolvasása közben ki tudjuk választani a szeparátort és a decimális elválasztót, erre mindig különösen figyeljünk oda, mert egyéb esetben egy oszlopban lesznek az adatok. (példa: `STATS.csv`)



- quote ’ jel a szövegben

  Más a különböző függvények quote alapbeállítása (példa: movie_metadata_v2.csv)

  `read.table`: quote = ”\””

  `read.csv`: quote = ”\””

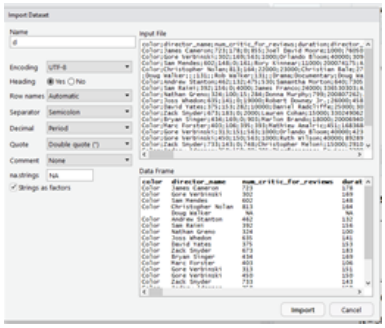
  ```
  d <- read.table("movie_metadata_v2.csv", header=T, sep=";")
  ```

  ```
  Error in scan(file = file, what = what, sep = sep, quote = quote, dec = dec,
  :  line 9 did not have 30 elements
  ```

  ```
  d <- read.csv("movie_metadata_v2.csv", sep=";")
  ```

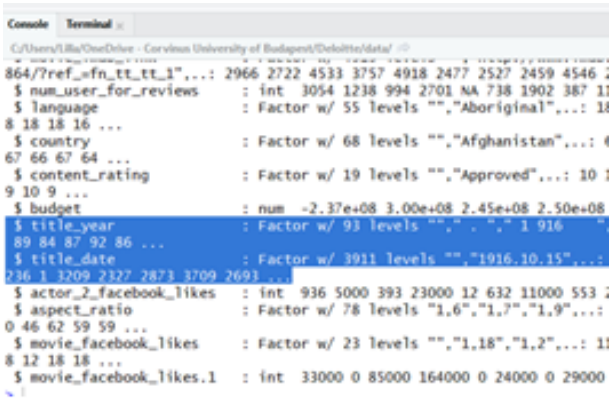
- Encoding

  Az adatok átnézése közben észrevehető, hogy számos fura karaktert tartalmaznak a szöveg változók, át kell állítani UTF8-ra a szövegek kódolását.



  ```
  d <- read.csv("movie_metadata_v2.csv", encoding="UTF-8", sep=";")
  ```

  A fájl „sikeres” beolvasása után, a `str(d)`, `summary(d)` kódok segítségével meggyőződhetünk róla, hogy várhatóan sohasem „sikeres” az adatbeolvasás.



10.2. Típusfelismerési és típuskonverziós hibák

- Faktor, dátum

  A szöveget tartalmazó kategória változók tárolására az R a faktor változókat alkalmazza, azaz egy kódtáblában annyi számot rendel hozzá, ahány különböző érték szerepel az adott változóban. Ennek köszönhetően a statisztikai modellek tudják kezelni ezen változókat is.

  Hátránya:

  Pl: Az adathibát (pont szerepel az évszámok között), nem megfelelő formátumban lévő változókat (pl. dátum) faktornak olvassa be, és ezeken módosítanunk kell.

  ```
  d$title_year<-as.numeric(as.character(d$title_year))
  ```

  ```
  d$title_date<-as.Date.character(d$title_date, format="%Y.%m.%d")
  ```

10.3. Hiányzó adatok kezelése

- Az adatbeolvasás során az üres helyeket `NA` értékekkel töltötte fel az R.

  Ez egy egyszerű átlag függvény használatát is megnehezíti, és torzíthatja az elemzésünket.

  ```
  mean(d$num_critic_for_reviews)
  ```

  helyett

  ```
  mean(d$num_critic_for_reviews, na.rm=TRUE)
  ```

  Megoldási lehetőségek (szakértői döntés alapján kell mérlegelni):

- cseréljük ki nullákra az `NA` értékeket (ez megváltoztathatja a leíró statisztika értékeit)!

  ```
  d$num_critic_for_reviews[is.na(d$num_critic_for_reviews)] <- 0
  ```

- cseréljük ki a várható értékre az NA értékeket (ez megváltoztathatja a leíró statisztika értékeit)!

  ```
  d$num_critic_for_reviews[is.na(d$num_critic_for_reviews)]
  <-mean(d$num_critic_for_reviews, na.rm=TRUE)
  ```

- hozzunk létre egy új táblát, ami nem tartalmaz NA értékeket (nagymértékű adatvesztéssel járhat, jelen esetben számos olyan változó van, amelyet nem fogunk használni, de hiányos az adott érték és így el fogjuk veszíteni ezeket a sorokat pl: content rating)!

  NA-t nem tartalmazó tábla

  ```
  dujtabla<-d[complete.cases(d), ]
  ```

10.4. Adathibák kiszűrése

10.4.1. Numerikus változókban

Következő lépésben át kell nézni minden numerikus változó leíró statisztikáját, hogy található-e a táblában adathiba.

`summary(dujtabla)`

Duration – filmek hossza változón keresztüli bemutatás

```
    duration
Min.   :-4000.0
1st Qu.:   95.0
Median :  106.0
Mean   :  165.5
3rd Qu.:  120.0
Max.   :72000.0
```

1. Negatív hosszúságú biztosan nem lehet egy film sem, ezek az értékek hibásak (újra a 10.3 alfejezetben ismertetett kérdések merülnek fel: töröljük az egész sort, vagy cseréljük ki más értékre).

   ```
   dujtabla[dujtabla$duration<=0,"duration"]<-mean(dujtabla$duration)
   ```

2. A duration változó sűrűségfüggvénye alapján megállapíthatjuk, hogy túl magas értékek is találhatóak.

   ```
   plot(density(dujtabla$duration))
   hist(dujtabla$duration)
   ```

   Milyen hosszú film számít extrém soknak? Adatrögzítési hiba? (szakértői véleményre van szükség) Szakértői vélemény: 10 óránál nincs hosszabb film: 10*60=600 perc feletti értékek rosszak `summary(dujtabla$duration)`

```
dujtabla[dujtabla$duration>600, "duration"] <- mean(dujtabla$duration)
hist(dujtabla$duration)
```

3. Hibás algoritmus javítása

   Az 1. lépésben helyettesítettük az átlaggal a negatív értéket, majd ezt követően megállapítottuk, hogy sok extrém magas érték van, tehát az extrém magas értékeket tartalmazó átlaggal való behelyettesítés hibás döntés volt. Célszerű lett volna NA-t helyettesíteni minden hibás adat helyére, és utána az összes NA-t együtt kezelni.

```
dujtabla<-d[complete.cases(d), ]
dujtabla[dujtabla$duration>600 | dujtabla$duration<=0,]<-NA
hist(dujtabla$duration)
dujtabla$duration[is.na(dujtabla$duration)] <-
mean(dujtabla$duration, na.rm=TRUE)
```

10.4.2. Szöveget/Faktorokat tartalmazó változókban

- A nyelv és az országok esetében is észrevehető az összes faktor kilistázásával, hogy találhatóak hibás adatok.

```
summary(dujtabla$language)
```

vagy

```
table(dujtabla$language)
```

```
> summary(dujtabla$language)
      Aboriginal          Arabic          Aramaic          Bosnian        Cantonese          Chinese 
               3               2               1                1                1                8 
       Chineseee        Chinesse           Czech           Danish            Dari            Dutch 
               0               0               1                3                2                3 
        Dzongkha             Eng         Engglish         Engliish          English       Englishhh 
               1               2               1                1             3683                1 
         Engllish        Filipino          French          German           Greek          Hebrew 
               1               1               1               13               0                2 
           Hindi       Hungarian       Icelandic      Indonesian         Italian        Japanese 
              10               1               1                2                7               12 
         Kannada          Kazakh          Korean         Mandarin            Maya        Mongolian 
               0               1               5               15                1                1 
            None       Norwegian         Panjabi         Persian          Polish      Portuguese 
               1               4               0                3                0                5 
        Romanian         Russian        Slovenian         Spanish         Swahili         Swedish 
               1               1               0               26                0                1 
           Tamil           Telugu            Thai            Urdu      Vietnamese            Zulu 
               0               1               3                0                1                1 
```

```
dujtabla$language[dujtabla$language == 'Eng'] <- 'English'
dujtabla$language[dujtabla$language == 'Engglish'] <- 'English'
dujtabla$language[dujtabla$language == 'Engliish'] <- 'English'
dujtabla$language[dujtabla$language == 'Englishhh'] <- 'English'
dujtabla$language[dujtabla$language == 'Engllish'] <- 'English'
```

```
summary(dujtabla$language)
```

Továbbra is láthatók ezek a kategóriák, tehát törölni is szükséges őket.

```
dujtabla <- droplevels(dujtabla)
summary(dujtabla$language)
```

10.5. Inkonzisztenciák az adattáblában

Az adattáblát jobban megismerve felfedezhetünk összeférhetetlenséget az adatokban. Ehhez szükséges, hogy már több napja az adattáblát vizsgáljuk.

Példa: Ethan Suplee két filmje között eltelt 77 év, várhatóan ez inkonzisztencia.

```
elteres<-aggregate(formula=title_year~actor_1_name,
data=dujtabla,FUN=function(x){max(x)-min(x)})
max(elteres[,2])
elteresszures<-elteres[elteres$title_year==77,]
dujtablateszt<-dujtabla[dujtabla$actor_1_name=="Ethan Suplee",]
```

Összegezve az adattáblában minimálisan végrehajtandó kódok:

```
d <- read.csv("movie_metadata_v2.csv", encoding="UTF-8", sep=";")
d$title_year<-as.numeric(as.character(d$title_year))
d$title_date<-as.Date.character(d$title_date,
format="%Y.%m.%d")
dujtabla<-d[complete.cases(d), ]
dujtabla$duration[dujtabla$duration>600 |
dujtabla$duration<=0]<-NA
dujtabla$duration[is.na(dujtabla$duration)] <- mean(dujtabla$duration, na.rm=TRUE)
dujtabla$language[dujtabla$language == 'Eng'] <- 'English'
dujtabla$language[dujtabla$language == 'Engglish'] <- 'English'
dujtabla$language[dujtabla$language == 'Engliish'] <- 'English'
dujtabla$language[dujtabla$language == 'Englishhh'] <- 'English'
dujtabla$language[dujtabla$language == 'Engllish'] <- 'English'
dujtabla <- droplevels(dujtabla)
```

Ezt a sort is érdemes törölni, vagy módosítani.

11. fejezet

Statisztikai elemzések R-ben

Az R – saját definíciója szerint – statisztikai modellezésre és vizualizációra kifejlesztett programozási nyelv. Az eszközkészlete az egyszerű lineáris regressziótól a neurális hálókig, gyakorlatilag minden adatelemzésre használt módszert lefed, és ami esetleg mégis hiányzik, csak idő kérdése és valamely újonnan fejlesztett package-ben elérhető lesz.

A következőkben – a teljesség igénye és legkisebb esélye nélkül – bemutatunk néhány egyszerű alkalmazást. Mivel szinte minden statisztikai eljárás előre megírt függvényekben le van programozva, a felhasználónak szinte korlátlan lehetőségek állnak a rendelkezésére, hogy akár modellek ezreit viszonylag egyszerű kódok megírásával maximális rugalmassággal alkalmazza a rendelkezésre álló adatokra. Ne felejtsük el azonban, hogy mindez nem pótolja ezen modellek ismeretét hiányát. Minden statisztikai modell kezelhető fekete dobozként, ha nem nekünk kell az algoritmust leprogramozni, azonban ez hatalmas felelőtlenség. A matematikai modellek feltételrendszerét és következtetéseit a felhasználónak pontosan ismernie kell, hogy valóban releváns következtetéseket tudjon levonni az adatok alapján.

A statisztikai modellek kívül esnek ennek a könyvnek a keretein, feltételezzük, hogy a hallgatók azokat korábbi tanulmányaikból már ismerik, ezért az R statisztikai alkalmazásait csak röviden, néhány egyszerű példafeladaton keresztül mutatjuk be.

11.1. Leíró statisztikák

11.1.1. Végezzük el a következő feladatokat:

- Olvassuk be a `movie_metadata.csv` fájlt! A fájl letölthető a `http://www.uni-corvinus.hu/~lkeresz` weboldalról (eredetileg a `https://www.kaggle.com/` website-ról származik)!
- Konzisztens-e a változók típusa a mérési szintjükkel?
- Készítsünk gyakoriság táblát a `language` változó értékeiből! Készítsünk egy táblát, ami az egyes

értékek előfordulási valószínűségét (relatív gyakoriság) tartalmazza. Az SPSS-ben a *frequencies*-ben található utasítás itt a `table` függvénnyel hívható meg, keressünk rá erre a help-ben.

- Ábrázoljuk diagramon a country változó gyakoriságait! Az x tengely feliratai legyenek függőlegesen olvashatók! (Kicsinyítsük le a betűtípust 25%-kal és növeljük meg a margót, hogy elférjenek a feliratok!) Ez nem olyan egyszerű, mert grafikus paraméterből irgalmatlan sok van, és a magas szintű függvények (pl. `plot`) help-jében nincsenek benne. Először keressünk rá a `?par` függvényre, azon belül a `xaxt` paraméterre, majd kutakodjunk tovább, amíg rá nem akadunk az `axis` függvényre!)

- Készítsünk boxplotot a duration és a `num_critic_for_reviews` változókból.

- Válasszuk ki az összes numerikus változót, és számítsuk ki a kovariancia és korrelációs mátrixot! Hogyan kezeljük a hiányzó értékeket?

- Készítsünk táblázatot, amelynek soraiban az egyes változók várható értéke, szórása, ferdesége, csúcsossága, és decilisei vannak, legfeljebb 3 tizedesjegyre kerekítve!

- Osszuk a megfigyeléseket 5 kategóriába a budget változó 20%-os kvantilisei szerint, és ábrázoljuk oszlopdiagramon kategóriánként egymás mellett a `movie_facebook_likes` és a `director_facebook_likes` változók átlagát. A kategóriákba soroláshoz egy *fkeres*-szerű függvényre lesz szükségünk, mely az R-ben `findInterval` névre hallgat. A kvantiliseket meglepő módon a `quantile` függvénnyel lehet megkeresni. A kategóránkénti átlag az Excelben a Pivot Table (vagy kimutatás) funkcióval valósítható meg, az SQL-ben pedig a GROUP BY utasítással. Ugyanezt a funkciót ellátó függvény az R-ben az `aggregate`.

11.1.2. Megoldások

- Az adatok beolvashatóak például a `d <- read.csv("movie_metadata.csv",header=TRUE, encoding = "UTF-8")` utasítással, vagy a sokoldalúbban paraméterezhető `read.table` függvénnyel.

- Az adatok beolvasása után mindig érdemes az `str` függvénnyel megvizsgálni a tábla szerkezetét. Itt látszik a megfigyelések száma, valamint az összes változó és típusa. A statisztikában a változóknak általában három vagy négy különböző mérési szintjét szokás megkülönböztetni: kategóriaváltozó, ordinális, intervallum szintű és arányskála, az utóbbi kettő között nem túl éles a határvonal. Az `str` meghívásával célszerű ellenőrizni, hogy a változók típusa ezzel konzisztens-e. Noha nincs áthághatatlan szabály arra nézve, hogy milyen mérési szintű változó milyen típusú legyen, a kategóriaváltozókat például célszerű `factor` változótípusban tárolni. Ez készít egy listát az összes lehetséges kategóriáról, majd a megfigyeléseket beszámozza aszerint, hogy hányadikak a listában. A `factor` változótípus használatával bizonyos hibákat ki lehet

küszöbölni, bizonyos hibákat pedig el lehet követni, melyek egyébként nem lennének. Előnyei és hátrányai is vannak tehát.

- Az előfordulási gyakoriságokat a `table(d$language)`, a relatív gyakoriságokat pedig a `prop.table(table(d$language))` utasítással lehet legegyszerűbben előállítani.
- Egy korrektnek mondható kinézetű ábrát állít elő a következő kód:

```
windows(20,10);par(mar = c(10,4,4,4), cex = 0.75)
countryFreq <- table(d$country)
plot(countryFreq, xaxt = "n" )
axis(1,at = 1:length(countryFreq), labels = names(countryFreq), las = 2 )
```

- `boxplot(d[,c("duration","num_critic_for_reviews")])`
- Ha megfelelően állítottuk be az adatok típusát, akkor az intervallum szintű változókat arról lehet felismerni, hogy `numeric` vagy esetleg `integer` típusként vannak tárolva. Így egy `for` ciklussal végigmenve a tábla oszlopain kiválogathatjuk azokat, melyek ilyenek:

```
NumValts <- NULL
for (i in 1:ncol(d)){
    if (class(d[,i]) == "integer" | class(d[,i]) == "numeric"){
        NumValts <- c(NumValts,i)
    }
}
d_num <- d[,NumValts]
```

  A korrelációs-, illetve kovarianciamátrix szimplán a `cor`, illetve `cov` függvények meghívásával számolható ki. A hiányzó értékek kezelésére azonban több lehetőség van, melyek közti választás nagyon nem triviális. Ezzel kapcsolatban célszerű elolvasni például a `?cor` help-ben p a `use` változó lehetséges beállításait. Ha nincs túl sok hiányzó érték, és a mátrixokat biztosan pozitív semidefinitnek akarjuk megkapni, akkor a

```
cor(d_num, use = "complete.obs")
cov(d_num, use = "complete.obs")
```

  beállítás ajánlott.

- A feladat megoldható nagyon bonyolult módon is, de viszonylag egyszerűen is, például az `apply` függvény segítségével. Ezzel (`for` ciklust helyettesítve) alkalmazhatunk egy függvényt egy vektor elemeire, egy mátrix vagy `data.frame` soraira vagy oszlopaira. Ha a függvény egyetlen

értéket ad vissza, oszlopokra alkalmazva egy olyan hosszú vektor keletkezik, ahány oszlopa volt a `data.frame`-nek. Ha a függvény maga is vektort ad vissza, mely minden lehetséges inputra ugyanolyan hosszú, akkor az eredmény egy mátrixba rendeződik az alábbi példánál is. A mátrix *i*-edik sorának *j*-edik eleme a függvény által visszaadott *i*-edik érték a *j*-edik oszlopra alkalmazva. Tehát, ha a változókat a sorokba akarjuk tenni, akkor a végén még transzponálni kell:

```
tabla <- apply(d_num,2,
         function(x){
              c(
                   mean(x,na.rm = TRUE),
                   sd(x, na.rm = TRUE),
                   mean( scale(x)^3, na.rm = TRUE),
                   mean( scale(x)^4, na.rm = TRUE),
                   quantile(x,1:9/10, na.rm = TRUE)
              )
         }
     )
```

Ezzel össze is állítottuk az adatot, amire szükségünk van, a többi már csak dekoráció. Transzponáljuk a táblát és beállítjuk a hiányzó fejléceket:

```
tabla <- t(tabla)
colnames(tabla)[1:4] <- c("mean","sd","skewness","kurtosis")
```

Végül szépen megformázva idézőjelek nélkül kiírjuk:

```
noquote(format(round(tabla,3), scientific = FALSE))
```

- Ha szorgalmasan végigbogarásztuk a help-ben a felhasználandó függvények leírását, egyszerű lépésekben eljuthatunk például a következő megoldáshoz:

  Az egyszerűség kedvéért válasszuk ki egy külön táblába azokat a változókat, melyekre szükségünk lesz:

  ```
  d1 <- d[,c("movie_facebook_likes","director_facebook_likes","budget")]
  ```

  Nem kötelező, de megkönnyíti az életünket, ha megszabadulunk minden hiányzó értéktől. Erre több mód is van az `is.na` függvény használatával, de használhatunk egy külön erre a célra kitalált `complete.cases` függvényt:

  ```
  d1 <- d1[complete.cases(d1),]
  ```

A budget változó kvantiliseit tartalmazó vektor:

```
q <- quantile(d1$budget,1:4/5)
```

Létrehozunk egy külön vektort, mely minden egyes megfigyelésről megmondja, melyik budget szerinti kategóriába tartozik:

```
kat <- findInterval(d1$budget,q)
```

A kategóriánkénti átlagot egyszerre több változóra is elkészíthetjük és az R segítőkészen egy mátrixba rendezi ezeket:

```
atlagok <- aggregate(d1[,1:2], by = list(kat), mean)
```

Természetesen a barplot függvény pont nem ebben a formában várja a megjelenítendő adatot, hanem fordítva, így a végeredményt transzponálni kell, mielőtt átadjuk neki, de ezek már csak technikai apróságok, melyekre a help-ben többnyire vannak példák.

```
barplot(t(as.matrix(atlagok[,2:3])), beside = TRUE)
```

11.2. Hipotézisvizsgálat

Hipotézisvizsgálatról R-ben kevés okos dolgot lehet mondani. Megjegyezhetetlen mennyiségű (de legjobb barátunk, a Google segítségével könnyen megtalálható) teszt van beépítve különböző packageekben, melyek kiszámolják nekünk a tesztstatisztika értékét és a p-értéket. Ez utóbbi néha analitikusan, néha szimulációval történik, függően a tesztstatisztika nullhipotézis melletti eloszlásától (mely bizonyos esetekben analitikusan kezelhetetlen).

Ennél sokkal többre nincs is szükség (illetve ennél többet egy géptől nemigen várhatunk), az eredmények értelmezése már a felhasználó felelőssége. Ezzel kapcsolatban jegyezzük meg, hogy a statisztikai próbák (más szoftverekhez hasonlóan) R-ben sem tesznek semmiféle erőfeszítést a próba alkalmazhatósági feltételeinek ellenőrzésére, sőt általában nagy fokú rugalmasságot biztosítanak a felhasználónak, hogy bármit kiszámoljon, amit szeretne.

Illusztrációként végezzük el a fejezet elején beolvasott filmek adatait tartalmazó táblán (legyen a tábla neve movies) a következő feladatokat!

- Bontsuk a imdb_score változót két almintára aszerint, hogy a movie_facebook_likes változó értéke kisebb-e, mint 3330 vagy nagyobb!
- Hasonlítsuk össze a két almintát! Döntsük el statisztikai próbával, hogy egyenlőek-e az átlagaik!

A két almintát *x*-szel és *y*-nal jelölve:

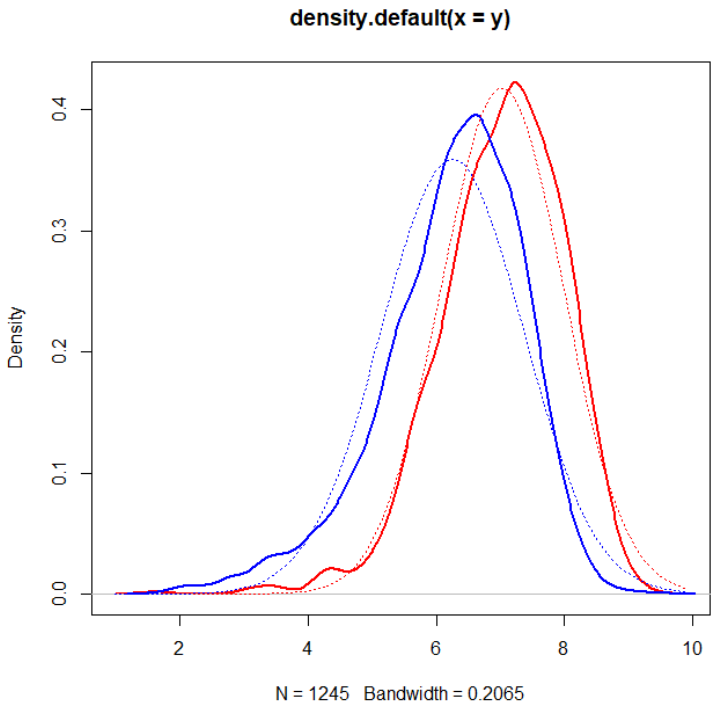
```
x <- movie$imdb_score[movie$movie_facebook_likes < 3330]
y <- movie$imdb_score[movie$movie_facebook_likes >= 3330]
```

A várható értékek összehasonlítására szolgáló t-próba elvégezhető a `t.test(x,y)` utasítás beírásával. Ha az eredményt egy változóba is elmentjük: `teszt <- t.test(x,y)`, akkor létrejön egy objektum (típusát tekintve egy lista), mely tartalmazza a teszt eredményeit, így azokat később el tudjuk érni és fel tudjuk használni például a `teszt$p.value` változón keresztül. A teszt eredményeként visszakapjuk a t-próba értékét, a hozzátartozó p-értéket, az x és y változók mintából becsült átlagát, és még néhány kevésbé érdekes apróságot, például a szabadsági fokot, tehát minden olyan numerikus információt, melyre szükségünk lehet.

Ez azonban csak a jéghegy csúcsa. Ha rákeresünk a help-ben a `?t.test` függvényre, akkor láthatjuk, hogy mely paramétereket lehetne elvileg még beállítani. Például egy- vagy kétoldali ellenhipotézist szeretnénk, feltételezzük-e, hogy a két mintában a szórások megegyeznek, stb. Ezeknek a paramétereknek a jelentésével a felhasználónak tisztában kell lennie és a teszt elvégzésekor tudnia kell, hogyan adja meg őket. Például, ha az adatgyűjtés módszertanából/technológiájából következően a két mintában a szórásoknak meg kell egyezni, akkor ezt kell specifikálni a teszt elvégzésekor. Ha nem, akkor külön F-próba végezhető a szóráshomogenitásra és annak eredményét figyelembe véve lehet megadni ennek a tesztnek ezt a paramétert. Ne felejtsük el, hogy ezen beállításoktól függően a tesztnek más-más nullhipotézise lehet (azaz más-más hipotézist tesztelünk) és a tesztstatisztika eloszlása (legalábbis a szabadságfok) is különbözhet. Így ugyanahhoz a tesztstatisztikához más-más p-érték tartozhat.

Ezeken felül tisztában kell lennünk azzal is, hogy a tesztek elvégzésének matematikai feltételei is vannak, valamint azzal is, hogy ezek a feltételek általában nem teljesülnek, és azzal is, hogy ez esetben melyik feltételre mennyire robosztus az adott teszt (azaz mekkora problémát okoz, ha elhanyagoljuk). A t-próba esetén például mindkét mintának normális eloszlásúnak kell lennie. Hogy lássuk ennek a teljesülését, ábrázoljuk a két minta sűrűségfüggvényét, a külön-külön rájuk illesztett normális eloszlással együtt:

```
plot(density(y), col = 2, lwd = 2);
lines(density(x), col = 4, lwd = 2)
mu1 <- mean(y); s1 <- sd(y); mu2 <- mean(x); s2 <- sd(x)
curve(dnorm(x, mu1, s1), col = 2, lty = 3, add = TRUE, lwd = 1)
curve(dnorm(x, mu2, s2), col = 4, lty = 3, add = TRUE, lwd = 1)
```



Az ábrán láthatjuk, hogy a várható értékek különbözőnek tűnnek, azonban az eloszlásokkal kapcsolatban egyáltalán nem egyértelmű, hogy mekkora hibát követtünk el, ha feltételezzük a normalitást. Egy további támpontot adhat, ha elvégzünk egy normalitástesztet is. Ebből több is rendelkezésre áll az R-ben, használhatjuk például a legáltalánosabb Kolmogorov-Szmirnov tesztet (ez alapértelmezetten rendelkezésre áll a `stats` package-ben, vagy a mostanában népszerű Jarque-Bera normalitástesztet (ez a `tseries` package-ben található).

A `ks.test(scale(x), pnorm)` kód elvégzi a kívánt műveletet (és a p-érték szignifikáns), azonban warning-ot dob "`ties should not be present for the Kolmogorov-Szmirnov test`" üzenettel. Ennek értelmezéséhez tudnunk kell, hogy a KS-teszt folytonos eloszláshoz hasonlítja az adatsort, ami alapból nem hihető, ha ismétlődő értékeink vannak, ezért kapjuk a figyelmeztetést. A JB-teszt ilyen üzenetet nem generál:

```
library(tseries)
jarque.bera.test(y)
```

azonban ha ez a teszt is elbukik, akkor a normalitás egyik adatsorról sem feltételezhető.

11.3. Többváltozós modellek, regresszió

Ahogy minden statisztikai programcsomagban, az R-ben is kész eljárások vannak a sokdimenziós statisztikai modellek paramétereinek becslésére a lineáris regressziótól a főkomponens-elemzésen át a neurális hálóig. Ezek használata relatíve egyszerű, alig valamivel nehézkesebb, mint az SPSS-hez ha-

sonló ”kattintgatós” szoftverek esetén.

Az előre programozott algoritmusok azonban csak számolni tudnak jobban nálunk, nem gondolkodnak helyettünk. Noha a legújabb szoftvereknek és a nagy számítási kapacitásnak köszönhetően a statisztikai modellek használatának határköltsége minimálisra csökkent az utóbbi évtizedekben, ez nem helyettesíti a modellek matematikai hátterének ismeretét. Mivel ennek a jegyzetnek a keretében nem kívánunk elmerülni a többváltozós statisztika rejtelmeiben, csak egy egyszerű példán keresztül illusztráljuk a modellek használatának technikáját:

- Illesszünk lineáris (egyváltozós) regressziót a `movie_facebook_likes` (célváltozó) és a `director_facebook_likes` (magyarázó változó) változókra! (Gyalázatos módon, most ne foglalkozzunk azzal a problémával, hogy miért lenne a kettő között lineáris kapcsolat.)
- A regressziós modellt becsüljük meg külön-külön minden egyes évre, amelyikben legalább 100 megfigyelésünk van (használjuk a `titleyear` változót annak eldöntésére, hogy melyik évhez tartozik az adott film)!
- A modellek közül azokat tartsuk meg, amelyekben a magyarázó változó a t-próba szerint szignifikáns.
- Írjuk ki az eredményt egy táblázatba, melynek egyik oszlopában az évek vannak, másikban az R^2-ek. A táblázat legyen az R^2 szerint rendezve!

A modellek statisztikai hasznosságával (értelmével) most nem foglalkozunk, csak a technikát mutatjuk be.

Először is, meg kell vizsgálnunk a `title_year` változót: melyik évben, hány film készült. Majd kiválasztani azokat az éveket, amelyekhez legalább 100 film tartozik (a táblában minden film csak egyszer szerepel).

A változó egyes értékeinek gyakoriságait összesítő táblát a `table` függvénnyel állíthatjuk elő. Elegendő átadni a vizsgálni kívánt változót, melyet az egyszerűség kedvéért nevezzünk el x-nek, hogy ne kelljen mindig a teljes változónévre hivatkozni:

```
x <- movie$title_year
a <- table(x)
```

Az `a` változó most egy vektor, melynek elemei az egyes évek gyakoriságai (hányszor szerepelnek az adatsorban), az előforduló értékek (azaz évek) a ”fejlécben”, azaz a vektor `names` attribútumában vannak.

Nincs más dolgunk, mint leszűrni az `a` változót azon értékeire, melyek elérik a 100-at. Ezután a `names` attribútumból ki kell szedni, hogy mely évekhez tartoznak ezek az értékek, végül - mivel a `names` attribútomot az R formálisan string-ként kezeli, vissza kell alakítani számmá:

```
v <- as.integer(names(a[a>=100]))
```

A `v` vektor most már tartalmazza az összes olyan évszámot, amelyikben legalább 100 filmünk van. Ha le akarjuk szűrni az egész táblát ezen adatokra, akkor az a `movie[movie$title_year %in% v,]` utasítással tehetjük meg (de erre nincs is feltétlenül szükségünk).

A feladat megoldásához most a következőket célszerű tenni:

- Végigiterálunk egy ciklussal az éveket tartalmazó `v` vektoron, és leszűrjük a táblából a kiválasztandó megfigyeléseket.
- A ciklus belsejében futtatunk egy-egy lineáris regressziót.
- Kikeressük az output-ból a t-statisztika p-értékét; ha ez szignifikáns, akkor elmentjük a modell R^2-ét egy külön táblába a hozzá tartozó évszámmal együtt.

Ahhoz, hogy ezt meg tudjuk csinálni két dolgot kell tudni:

- Hogyan futtassunk R-ben regressziót? (Ez a könnyű része.)
- Hogyan szedjük ki a regresszió eredményét, ha nem egyesével szeretnénk manuálisan kiválasztani a képernyőről? (Ez egy kicsit trükkösebb, de épp ez a leghasznosabb az egészben.)

Lineáris regressziót futtatni R-ben a `glm`, vagy az `lm` függvénnyel lehet. A `glm` (General Linear Model) egyéb lineáris modelleket is tud, például logit regressziót, az `lm` függvény kifejezetten logisztikus regressziót futtat. A függvény paraméterezése úgy történik, hogy át kell adni neki az összes adatot egy `data.frame` formátumban, és ki kell jelölni a célváltozót és a magyarázó változókat. A változók kijelölése a `formula` paraméter átadásával történik, melynek alakja:

$$celvaltozo \sim magyarazovaltozo_1 + magyarazovaltozo_2 + \cdots + magyarazovaltozo_k$$

A formulák fenti alakja teljesen általános. Függetlenül attól, hogy lineáris vagy logit regressziót, döntési fát, vagy éppen neurális hálót akarunk futtatni, minden olyan statisztikai eljárás esetén, mely egy célváltozót és több magyarázó változót tartalmaz, a modellt ugyanígy kell specifikálni. Át kell adni a függvénynek egy string-et, melyet nem kötelező idézőjelek közé tenni, és először tartalmazza a célváltozót, utána egy tilde karaktert, majd az összes magyarázó változót összeadásjelekkel elválasztva. Az összeadásjeleknek itt nincs matematikai tartalma vagy funkciója, pusztán a magyarázó változók egymástól való elválasztására szolgál. Mind a célváltozónak, mind a magyarázó változóknak az átadott adattáblában lévő fejlécekkel kell pontosan megegyeznie. Ebben a formában az R felismeri, hogy mely változókra és mely adatokra kívánjuk a modellt futtatni és – ha valamilyen váratlan hibába nem ütközik, például szinguláris kovarianciamátrix – akkor lefuttatja a modellt. Például, ha a lineáris regressziót a teljes adattáblára szeretnénk futtatni, akkor ezt az

```
lm(movie_facebook_likes~director_facebook_likes, data = movie)
```

utasítással tehetnénk meg.

Egy kicsit nehezebb feladat a regresszió eredményének felhasználása. Az ugyanis nem lenne túl hatékony, ha a program az outputokat egyszerűen kiprintelné, hiszen, akkor egyesével kellene őket végigbogarászni, aztán az eredménnyel kezdeni valamit. Az R lehetővé teszi, hogy egy probléma kapcsán akár több ezer lehetséges modellt ráillesszünk egy adatsorra, és ezek közül automatizáltan szelektáljunk előre adott, objektív kritériumok alapján. Ez úgy valósul meg, hogy a modellek outputjai egy külön változóba kerülnek, melyből aztán a változó nevén keresztül hozzáférhetőek. Ez a változó egy lista típusú objektum, így az egyes adattagjai bármilyen más típusú objektumokat (mátrixokat, vektorokat) tartalmazhatnak, melyekhez a $ operátorral tudunk hozzáférni. Ehhez a modellt úgy kell lefuttatni, hogy eredményét értékül adjuk egy változónak:

```
modell <- lm(movie_facebook_likes~director_facebook_likes, data = movie)
```

Az így létrejött változóból például a `modell$coefficients` utasítással szedhetjük ki a bétákat (regressziós együtthatókat), mely egy elemi vektor, hossza a magyarázó változók száma $+1$ a konstans. Hasonlóan a `modell$residuals` tartalmazza a hibatagokat, ez egy olyan hosszú vektor, ahány megfigyelésünk van.

Ha az `str(modell)` utasítással végigbogarásszuk a kapott objektumot, úgy tűnik, hogy sem a t-statisztikák értékeit, sem az R^2-et nem tartalmazza, amire szükségünk lenne. A `summary` utasítás egy újabb objektumot hoz létre, melyben megtalálhatóak ezek az adatok, és még sok minden más. Azt mondhatjuk tehát, hogy a modellek outputjai olyan bonyolult szerkezetű objektumok, melyekből a minket érdeklő adatokat "ki kell kukázni" és külön elmenteni valahova. Ez a kis kellemetlenség az ára annak, hogy egyszerre akár sok ezer modell közül választhattuk ki a legjobbat.

Ha létrehozzuk az `osszesites <- summary(modell)` változót, akkor ebben már megtalájuk az R^2-et az `osszesites$r.squared` tagban. Az együtthatók statisztikáit a `osszesites$coefficients` tábla tartalmazza, mely egy négy oszlopból álló mátrix, és annyi sora van, ahány együttható (a konstanst is beleértve). Az utolsó két oszlop tartalmazza a t-próba értékét és a p-értéket. Ha tehát a sorban az i-edik magyarázó változó t-statisztikájára, illetve ennek p-értékére vagyunk kíváncsiak, akkor ezt a `osszesites$coefficients[i+1,3]` illetve a `osszesites$coefficients[i+1,4]` változókon keresztül érhetjük el.

Mindezek után már viszonylag könnyen megoldhatjuk a feladatot. Az éveknek megfelelően futtatunk egy ciklust. Minden ciklus belsejében létrehozzuk a regressziós modellt, és az ennek `summary`-jét tartalmazó objektumot. A summary-ből kiszedjük az egyetlen magyarázó változónk t-próbájának p-értékét, mely mindig a `coefficients` tábla második sorának negyedik oszlopában lesz. Amennyiben ez az érték szignifikáns, az adott évet és a modell R^2-ét kimásoljuk egy erre létrehozott táblába. A táblát végül rendezhetjük az R^2-ek szerint (eredetileg az évek alapján lesz rendezve), mert az a fontosabb adat.

Felhasználva a korábban létrehozott v változót, mely az éveket tartalmazza, csupán az alábbi egyszerű ciklust kell megírni:

```
Tablazat <- NULL
for (i in v){
  Leszurt_tabla <- movies[movies$title_year == i,]
  modell <- lm(movie_facebook_likes~director_facebook_likes,
                                                    data = Leszurt_tabla)

  osszesites <- summary(modell)
  if (osszesites$coefficients[2,4] < 0.05){
    Tablazat <- rbind(tablazat,c(i,osszesites$r.squared))
  }
}
```

Ezek után már csak meg kell formázni a táblázatot, hogy kicsit szebben nézzen ki:

```
colnames(Tablazat) <- c("Év", "R-squared")
Tablazat <- Tablazat[order(Tablazat[,2]),]
```

12. fejezet

Záró gondolatok

Becslések szerint az emberiség jelenleg évente több adatot generál és tárol adathordozókon, mint amennyit a történelem kezdetétől a legutolsó ezredfordulóig összegyűjtött. Noha ezen adatoknak valószínűleg elenyészően kicsi töredéke tartalmaz hasznos információt, a cégek egyre nagyobb erőforrásokat fordítanak a bennük lévő esetleges üzleti haszon megszerzésére. Emellett a tudományos kutatás is nagymértékben támaszkodik adatelemzésre. Az adatok feldolgozásának, elemzésének, vizualizációjának és interpretálásának képessége egyre növekvő piaci érték. Mindemellett a problémák számítógéppel való megoldása a technológia fejlődésével és terjedésével olyan képességgé kezd válni, mint a középkorban az írástudás: kezdetben hatalmas előny, majd egyre nélkülözhetetlenebb.

Az R az egyik legkönnyebben és leggyorsabban megtanulható és ma már nagyon könnyen hozzáférhető programozási nyelv. Amit ebben a jegyzetben szerettünk volna bemutatni, kevesebb, mint a jéghegy csúcsának a csúcsa, azonban arra talán elegendő, hogy a hallgató (olvasó) megtehesse az első lépéseket a nyelv megtanulásának és hatékony felhasználásának irányába. Aki szeretne az adatelemzés, modellezés és programozás területén további tudásra szert tenni, annak szerencsére számtalan forrás áll rendelkezésére. Kezdetnek ajánljuk az r-project.org weboldalon lévő, hivatalos, bevezető R jegyzetet: `https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf`. Egy lényegesen technikaibb, elsősorban haladó programozóknak ajánlható irodalom az R nyelv absztrakt definícióját tartalmazó dokumentáció: `https://cran.r-project.org/doc/manuals/r-release/R-lang.pdf`. A statisztikai modellek matematikai háttere iránt érdeklődőknek leginkább Ramanathan Statistical Methods in Econometrics című könyvét ajánljuk.

Az R-project.org weboldalon további források tucatjait lehet fellelni. A 24 óra alatt mindent megtanító könyvek és a lineáris regressziótól a machine-learningig minden témát összesen 150 oldalban lefedő, nagyon felszínes tudást adó gyorstalpalókat jó szívvel nem ajánljuk. Ne felejtsük el, hogy a valószínűségszámítás, a statisztika és a programozás is tudomány, melyek elsajátításához idő és szorgalom kell.