Debreceni Egyetem

Informatikai Kar

Csomagfejlesztés az R környezetben

Témavezető:

Jeszenszky Péter

tanársegéd

Készítette:

Kovács Tibor Krisztián

III. programtervező informatikus

Debrecen

2008

Tartalomjegyzék

Előszó

A dolgozatom célja hogy az R programozási nyelvet egy rövid, egyszerű és érthető bevezetéssel bemutassam alapfokon. Egy csomag fejlesztésével felvázoljam azok szerkezetét és felépítését, így segítséget nyújthassak azoknak, akik az R-ben szeretnének elkezdeni fejleszteni. Megismerkedünk majd a könyvtárszerkezettel, az azokban kötelező, valamint használható fájlokkal, és a dokumentálással – ami nagyon fontos az R-ben annak használhatóságára nézve. A dolgozatomban a neurális hálók egyik speciális fajtáját megvalósító csomagot fogok fejleszteni. Így definiálom a Hopfield hálókat, és leírom azok reprezentációjának részleteit R-ben. A neurális hálók témakört, ami egyébként elég nagy, egy minimális bevezető erejéig érintem, hogy a bemutatott Hopfield hálókat könnyebben tudjuk értelmezni.

1. Bevezetés

1.1. Mi az R?

Az R egy programozási nyelv és környezet, amely kiválóan alkalmas statisztikai számítások, adatokon végzett manipulációk és grafikus megjelenítések elvégzésére. Az R annak a John Chambers és kollégái által fejlesztett S nyelvnek a GNU verziójaként is felfogható, amelyet az 1970-es években fejlesztettek a Bell Laboratories-ban, interaktív adatelemzés és vizualizáció céljából. A rokonság egyébként annyira nagy a két nyelv között, hogy az S-ben megírt kód változtatás nélkül, vagy kis átalakítással használható R-ben is.

A funkcionális programozási nyelvek családjába sorolható, magja lényegében egy parancsértelmező, ami a programkódot utasításonként értelmezi. Lehetővé teszi a moduláris programozást a függvényein keresztül. Valamint megvalósítható benne az objektumorientáltság is, habár nem a megszokott formában.

Sok okból népszerű, legfontosabb ezek közül az, hogy nyílt forrású, bár magját csak a core team fejleszti, az ötleteket szívesen meghallgatják. A már eddig is meglévő széleskörű funkcionalitással bíró csomagokat egyre többen bővítik, és teszik elérhetővé az R hivatalos honlapján, a CRAN-on, ahol rengeteg információt találhatunk nem csak a csomagokról, hanem segítséget kaphatunk a fejlesztéshez is. A honlapon többek közt olyan manuálok találhatóak, melyekkel megtudhatjuk az R installálását forráskódból, megtanulhatjuk az R kezelését, a csomagfejlesztést, az adat be- és kivitelt, sőt még az R magjának fejlesztéséhez is találunk leírást.

Fontos továbbá a hatékony adattárolást és kezelést lehetővé tévő eszközrendszer, mátrixokon és vektorokon való beépített műveletvégzés, valamint adatelemzéshez eszközrendszer és fejlett adatbevitel, kivitel. Ezeken kívül a csomagfejlesztés történhet teljesen más nyelven is, így függvényünket a sebességnövelés érdekében akár C-ben, C++-ban, esetleg Fortranban is megírhatjuk.

1.2. A neurális hálók

Biológiai értelemben a neurális hálózat alatt biológiai neuronok összekapcsolt csoportjára kell gondolunk, az idegrendszer funkcionálisan összefüggő területeire. Informatikai használatban a szó alatt a mesterséges neurális hálókat értjük, amelyek mesterséges neuronokból állnak, amik a biológiai neuronok egyszerűsített matematikai modelljeként jelentek meg. Későbbiekben, ezen tudományterület fejlődésével a modellek kezdtek elvonatkoztatni a kezdeti modellektől és váltak egyre absztraktabbakká.

A neuron lemodellezésére tett első kísérletek 1943-ban történtek, az első ilyen mesterséges neuron a Threshold Logic Unit (TLU) nevet kapta, megalkotója McCulloch és Pitts volt. A következő lépcsőfok a McCulloch-Pitts féle neuronra bevezetett tanuló eljárás volt 1958-ban, Rosenblatt perceptron fogalommal dolgozó modelljében a tanulás a várt és a kapott kimenet adta hiba visszacsatolásán alapult. 1960-ban Widrow és Hoff leírták az egykimenetű adaline (adaptive linear neuron) és a többkimenetű madaline modelljeiket. Ám a kutatásokban komoly törését jelentette Minsky 1969-es eredménye, miszerint egyetlen neuron nem képes a xor művelet leképezésére, a több neuronból álló modellhez pedig nem találtak megfelelő tanító algoritmust. Minsky eredményeinek hatására sokan hagyták abba a témában végzett fejlesztéseket.

A neurális hálók reneszánszának nagy mérföldköve volt a John Hopfield által kidolgozott asszociatív modell az 1980-as évek elején, amely egy egyrétegű, teljesen csatolt bipoláris háló volt. Hopfield megközelítése illusztrálta, hogy az elméleti fizikusok a számítási egységekre jobban szeretnek egységes egészként gondolni. Nem szükséges a szinkronizáció, minden egység elemi rendszerként működik, komplex kölcsönhatásban az egész többi részével. Az igazi áttörést a témában viszont a backpropagation tanulási eljárás jelentette 1986-ban. A legkisebb négyzetek elvén alapuló eljárás általánosítását többrétegű, nemlineáris átviteli függvényekkel rendelkező hálókra Rummelhat, Hinton és Williams írta le. Az új lendületet kapott témában neurális hálók sokasága jelent meg, valamint egyre többen kezdtek a témára alapuló programokat fejleszteni, felismervén használhatóságukat.

A modern neurális hálóknak három fontos tulajdonsága van: rendezett topológiájuk van, és rendelkeznek tanulási valamint az eltárolt információ előhívására használható

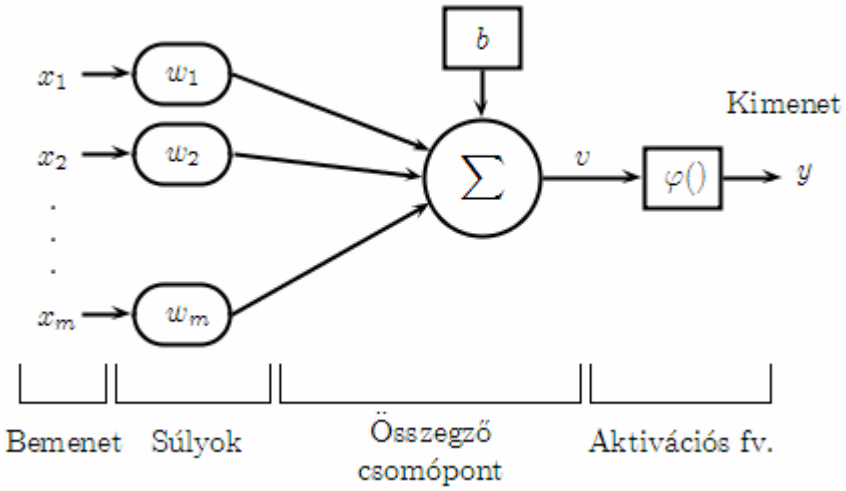
algoritmussal. Sok tanítási eljárás létezik, melyek segítségével a hálózat tulajdonságait dinamikusan változtathatjuk.

A neurális hálózatok jól használhatóak alak- és hangfelismerési, képfeldolgozási, osztályozási, approximációs és optimalizációs feladatok megoldására, illetve adattal címezhető és asszociatív memóriák valamint előrejelzések készítésére.

2. Neurális háló bevezetés

2.1. A perceptron

A mesterséges neurális hálózatok egységekből, azaz neuronokból épülnek föl. Egy neuron öt fő részből tevődik össze, ezek a következők: bemenet, súlyok, torzítás, összegző csomópont és az aktivációs függvény.



A bemenet egy n elemű valós X vektor, amely az aktuális bemeneti adatokat tartalmazza. Ezek az adatok vannak súlyozva egy W vektorral. Az összegző csomópontban összeadjuk a súlyozott bemeneti adatokat és hozzáadjuk a torzítást (biast, azaz b-t) a következő módon:

$$v = b + \sum_{i=1}^{n} w_i x_i$$

Az aktivációs függvény a problémának megfelelő tartományra képezi le az összegző csomópont által szolgáltatott eredményt. Bármely olyan eloszlásfüggvény, amely monoton, balról vagy jobbról folytonos valamint határértéke mínusz végtelenben nulla, plusz végtelenben pedig egy, az lehet aktivációs függvény. Néhány ilyen függvény:

$$\varphi(x) = \begin{cases} +1, & \text{ha } x > 0 \\ -1, & \text{ha } x \leq 0 \end{cases}$$

Lépcsősfüggvény

$$\varphi(x) = \frac{1}{1+\exp(-ax)}, \qquad \text{ahol } a > 0 \text{ konstans}$$

Logisztikus függvény

Az előzőekben említett tulajdonságú eloszlásfüggvényeken kívül léteznek egyéb speciális aktivációs függvények is.

A gerjesztettség a bemenetek súlyozott lineáris kombinációja, ezt jelöljük v-vel. A neuron működése egyszerűbben felírható a $y = \varphi(\sum_{i=0}^{n} w_i x_i) = \varphi(w^T x)$ alakban, amennyiben a nulladik bemeneti adatot egynek és a b torzítást nulladik súlynak vesszük, azaz $x_0 = 1$ és $w_0 = b$, ahol x a bemeneti vektor, w a hozzá tartozó súlyok vektora.

2.2. A topológia

A neuronok ugyanolyan vagy hasonló műveleteket végeznek, és ezt egy hálózatban a többitől függetlenül teszik, azaz párhuzamos feldolgozó egységek szerepét töltik be. Egy hálózatban a neuron sok másik neuronnal van összekapcsolva, és a mesterséges neurális hálók ezeknek a kapcsolatoknak a segítségével épülnek fel. A kapcsolatok topológiájával különböztetjük meg az egyes hálózattípusokat. A neuronok különböző rétegekbe vannak sorolva, a rétegben lévő egyes neuronok gerjesztettsége adja a réteg kimenetét. Az egyszerűbb hálózatokat legkönnyebben egy irányított gráffal írhatjuk le, ahol a neuronok a csomópontok, az irányok a kimenet felől a bemenet felé mutatnak. Ez a reprezentáció bonyolultabb esetekben átláthatatlanná válhat.

Egy neuron általában meghatározott számú másik neuronnal áll egyirányú kapcsolatban, ami alapján rétegekbe lehet sorolni őket. Az egyik réteg kimenete vagy az input lesz egy másik réteg bemenete. Három nagy csoportot szokás megkülönböztetni:

1. Input réteg: a bemeneti jelet továbbítják a hálózat felé.

2. Rejtett réteg: a feldolgozást végző neuronok tartoznak ide, ebből a rétegből több is lehet.

3. Output réteg: a külvilág felé továbbítják a rejtett réteg által kiszámolt információt.

A legtöbb hálózat esetén az egyes rétegben lévő neuronok kimenete a rákövetkező réteg összes neuronjának a bemenetével össze van kötve. A hálózatban a számítások

elvégzésének egyszerűbb módja, ha mátrixokkal reprezentáljuk az információkat. A bemenetet és a kimenetet elég egy-egy vektorral leírni. A l-edik réteg n darab neuronja és az $l+1$-edik réteg m darab neuronja között lévő súlyokat egy W mátrixba érdemes rendezni, ami egy $n \times m$ méretű mátrixot fog eredményezni. A W mátrix i-edik sorának j-edik eleme az l rétegbeli i-edik neuron kimenete és az $l+1$-edik rétegbeli j-edik neuron bemenete közötti súlyt fogja jelenteni. K db rejtett réteg esetén K db W mátrixunk lesz. Az olyan esetekben, amikor két neuron között esetleg nincs kapcsolat, ott a W mátrixba 0 kerül.

Lehetségesek különböző visszacsatolások a hálózatban. Ezek több féle esete is lehetséges. Lehetséges egy neuron kimenetét a saját bemenetére kötni, ez az elemi visszacsatolás, esetleg ugyanabban a rétegben lévő neuron bemenetére kötni, ami laterális visszacsatolást eredményez, vagy egy előző rétegben lévő neuron bemenetére kötni, ami rétegek közti visszacsatolást jelent.

2.3. Szinkron és aszinkron rendszerek

A neurális hálók tervezésének fontos problémája a neuronok megfelelő szinkronizációja. A McCulloch-Pitts hálók esetében ez az ütemezés megoldható, ha feltesszük, hogy az egy rétegben lévő neuronok aktivációja egységnyi időbe telik, azaz a bemeneti adatok megkapása és a kimenet kiszámítása egy időegységben történik. Ebben a felfogásban a neurális hálózat leírható egyszerű lineáris algebrai módszerekkel, vektorokkal és mátrixokkal. A hálózatot ennek a késésnek a figyelembevételével, valamint az egységek és a kapcsolatok megfelelő mintázatba rendezésével kell felépíteni.

A szinkronizációhoz fel kell tenni, hogy van egy külső óra, egy globális idő, amihez állítani kell az egységek számítását. Ám se a biológiában, sem a fizikában ilyen globális órát nem lehet feltételezni. Ez vezetett olyan modellekhez, melyekben elvetik a szinkronizációt. Az olyan neurális hálók sztohasztikus automatákként viselkednek, amelyekben a neuronok különböző időpontokban aktiválódnak, és a számítások elvégzésére az egységeknek kevés időre van szüksége. Az ilyen neuronokból felépített hálók sztohasztikus dinamikus rendszerekként viselkednek.

3. Az R és a neurális hálók

3.1. R alapok

Az R nyelv egy interpretált nyelv, így programjait nem fordítjuk le bináris állományba, hanem az R parancsértelmező értelmezi soronként. Ezt a terminált letölthetjük Unix platformokhoz (FreeBSD-hez is és Linuxhoz is), Windowshoz és MacOS-hoz. De ha olyan operációs rendszert használunk, ami nem volt a felsorolásban, nem kell megijednünk, mert a forráskódból saját magunk is lefordíthatjuk tetszőleges platformra a rendszert.

A kódok bevitelére két lehetőség van. Egyik a kód begépelése soronként az interpreterbe. A másik, hogy fájlba mentett scriptet töltünk be a `source()` paranccsal. Az R megkülönbözteti a kis valamint nagybetűket, így a `valami`, `Valami` és a `VALAMI` három különböző objektumot jelenthet. Az azonosítók nevében használhatóak az abc kis és nagybetűi, számjegyek, valamint a `.` és _ is, némi megszorításokkal. A szabályos azonosítók betűvel kezdődnek vagy ponttal. Viszont ha az azonosító ponttal kezdődik, akkor a második karakter nem lehet számjegy. A hordozható kódok írásához érdemes hanyagolni az ékezetes vagy egyéb nyelvspecifikus karaktereket. Megjegyzéseket a # után írhatunk. A megjegyzés egészen a sor végéig tart.

Többféle konstanst megadhatunk R-ben, ezek lehetnek logikai, numerikus, komplex és karakterlánc típusúak, valamint további három speciális, ezek a `NA`, `Inf` és a `Nan`. A karakterlánc típusú konstansokat `"` között vagy `'` között adhatunk meg. A numerikus konstansok a C konstansokra hasonlítanak. A logikai lehet `TRUE` vagy `FALSE`, míg a komplex `a-bi` alakú, ahol `a` a valós rész, `b` a képzetes. A valós rész elhagyható.

Fontosak viszont a speciális konstansok. Az `NA` hiányzó értéket jelöl, amely alapból egy logikai konstans, viszont át lehet konvertálni bármely típusra, így használhatjuk bármely objektumban hiányzó érték jelölésére. Műveletet végezve vele legtöbbször `NA`-t kapunk, kivéve ha a hiányzó adat nélkül is elvégezhető a művelet (például logikai műveletekben). Az `Inf` a végtelent jelenti, amelyet bármely matematikai függvényben használhatunk. A nullával való osztás `Inf` vagy `-Inf` értéket eredményez. Van még a `NaN` érték (Not A Number), amely egy valós konstans. Akkor kapunk `NaN` értéket, ha nullát osztunk nullával, vagy

végtelenből vonunk ki végtelent. Ez a konstans is megjelenhet matematikai függvényekben, valamint konvertálása esetén az eredményünk NA lesz.

Kifejezést, ha értékadó utasítás nélkül adunk meg az interpreternek, akkor az kiértékelődik, és az eredményt megjeleníti az R, ez az eredmény ekkor nem tárolódik el. Az értékadás általános szintaxisa `változó <- kifejezés`, ebben az esetben is kiértékelődik a kifejezés, viszont az eredmény objektumként a változóban tárolódik és nem jelenítődik meg.

Az utasítások egymás után értékelődnek ki. Az utasításokat pontosvesszővel vagy sortöréssel választhatjuk el egymástól. A pontosvessző mindig az utasítás végét jelenti, viszont ha sortörés volt, az utasítást csak akkor hajtja végre az interpreter, ha az szintaktikailag helyes. Ha nem az, akkor a prompt átvált interaktív módba, azaz megjelenik a + jel, és folytatnunk kell az utasítást. Egyéb esetben is interaktív módba kapcsolhatunk, például ha az utasításainkat blokkba szervezzük. Ezt a {} – kapcsos zárójellel – tehetjük meg, az R ugyanis addig nem értelmezi a kódot, amíg a blokkot be nem zárjuk. Tehát egy utasítás vagy magában áll, vagy blokkban szerepel.

Az R-ben létrehozott egységeket, változókat objektumoknak nevezik. Az R-ben tulajdonképp minden objektumként van kezelve. Minden objektumoknak van egy típusa, S nyelvel kompatibilis módja és tárolási módja.

Az R-ben 24 féle objektumtípus van. Vannak közöttük olyanok melyek fontosabbak, vannak, melyekkel a felhasználó sohasem találkozik, és vannak speciális objektumtípusok is. R-ben a legegyszerűbb objektum a vektor, amely azonos típusú objektumokból áll. Egy magában álló szám is vektor, csak egyelemű vektor. Egy vektornak öt típusa lehet, ezek a következőek: `logical`, `integer`, `double`, `complex`, `character`. Például a stringek `character` típusú vektorok. A listák olyan vektorok, amelyek elemei bármilyen típusúak lehetnek. A vektorok és listák elemeit a programnyelvekben megszokott módon a [] karakterek között indexelhetjük, a listák elemeire úgy is, hogy az objektumnév után $ jelet, valamint a hivatkozni kívánt komponens nevét írjuk. További objektum a kifejezés-objektum, amely szintaktikailag helyes, de nem kiértékelt kifejezéseket tartalmaz. Ezt többek között akkor érdemes használni, mikor egy matematikai formulát szeretnénk egy ábrán megjeleníteni.

R-ben fontos fogalom az attribútum, ezek már az egyszerű objektumoknál is megjelennek, csak kevesebb szerepet kapnak. Az attribútumok kulcs-érték párok és a NULL

objektum kivételével minden objektumhoz hozzárendelhetők, és különböző tulajdonságokat fognak jelenteni. Ezeket az `attributes()` függvénnyel tudjuk elérni. Az objektumok tulajdonképpen egy listaként vannak tárolva, melynek elemei az attribútumok is. Az objektum attribútumait írni az `attributes(obj)<-érték`-kel tudjuk, míg adott nevű attribútum értékéhez hozzáférni az `attr(obj,attrib.név)` függvénnyel lehet. Ezek az attribútumok a típusok megkülönböztetésére nagyon jól használhatóak, például a tömb típus a vektortól lényegében annyiban különbözik, hogy van két plusz attribútuma, a `dim` és a `dimnames`.

Összetett objektumok a faktorok és az adatkeretek. A faktorok olyan vektor jellegű objektumok, amelyek velük azonos hosszú vektorok elemeinek egy diszkrét osztályozását határozzák meg, mégpedig a `levels` attribútummal. Ez az attribútum kötelezően egy megfelelő elemszámú, csupa különböző karakterláncból álló karaktervektor kell hogy legyen, amely elemek fogják jelenteni a lehetséges kategóriákat. Míg az adatkeretek olyan speciális listák, amelyek elemei azonos hosszúságú objektumok. Úgy lehet rájuk gondolni, mint egy táblázatra, amelyben minden oszlop más-más típus lehet. Az összetevői lehetnek vektorok, listák, faktorok és mátrixok is, ahol a hosszal a sorok számának kell megfelelnie. Adatkereteknek van két attribútuma: a `names` karaktervektor, amelynek elemei az adatkeret oszlopainak neveit adják meg, és a `row.names`, ez is egy karaktervektor, amely a sorok nevét adja meg, nem lehet két egyforma érték és `NA` sem, ez tulajdonképp felfogható akár elsődleges kulcsként is (habár nem az, csak az indexelést szolgálja).

Az objektumokat manipulálni operátorokkal vagy függvényekkel lehet. Egy függvénynek a felépítése a következő: `függvény.név(`arg_1`, `arg_2`, …, `arg_n`)függvény.törzse.` Ez is egy objektum, és a függvénynévvel azonosítja a rendszer a meghívandó függvény(eket). Lehet változó argumentumszámú függvényt definiálni a három pont segítségével. A paraméterkiértékelés név és sorrendi kötés szerint történik az aktuális paraméterlista alapján. Ha van valamelyik argumentumnak alapértelmezett értéke, akkor az elhagyható, valamint az argumentum neve lerövidíthető, azaz elhagyhatunk a végéről karaktereket, addig, amíg az egyértelmű marad a sorrendet is figyelembe véve. A paraméterátadás pedig érték szerinti.

Az R munkafolyamat során létrehozott objektumok név szerint vannak tárolva. Ezeket a tárolt objektumokat együttesen munkaterületnek vagy környezetnek nevezzük, és tulajdonképpen szimbólumnév – érték párokból álló keret objektumok alkotják. Kiíratni őket

az `objects()` vagy `ls()` parancsokkal lehet, míg törölni a `rm(objektumnév)` parancsal lehet. A környezeteknek fa szerkezete van, és ezeket az R automatikusan kezeli, például minden függvényhívásnál létrehoz egy új környezetet, amely a függvény lokális változóit fogja tartalmazni. Környezetet explicit módon is létrehozhatunk.

Az R-ben az objektumorientáltságot egy speciális attribútum valamint a generikus függvények valósítják meg. Osztályokat létrehozhatunk a `class(objektum)` függvény segítségével, ami ha értékadásban szerepel, értéket ad az attribútumlistában lévő `class` attribútumnak, értékadás nélkül az objektum osztályát adja vissza. Az értékadásban egy karaktervektornak kell lennie, vagy lehet még `NULL` is, ebben az esetben töröljük az osztály attribútumot. Egy objektum több osztályhoz is tartozhat, azaz az osztály attribútum értéke nem feltétlenül egyelemű vektor. Amennyiben nincs explicit osztálynév az objektumhoz rendelve, akkor implicit módon származik az úgynevezett implicit osztályokból.

Egy osztályba tartozó objektumon a módszerek segítségével végezhetünk műveleteket, lényegében az R-ben ezek a módszerek a metódusoknak felelnek meg. Ezeknek a definíciója: `fun <- function(obj,…) UseMethod(funct)`, és az így létrehozott függvényeket fogjuk generikus függvénynek hívni. A függvény megállapítja az argumentumként kapott objektum osztályát, és az ahhoz az osztályhoz tartozó `UseMethod`-ban megadott függvényt hívja meg. Ennek a folyamatnak a neve a method dispatching. Egy generikus függvény hívásánál tetszőleges számú argumentumot megadhatunk, de egyet legalább meg kell, ami az az objektum lesz, amelyen meghívjuk a metódust. Amennyiben ennek az objektumnak van explicit osztálya és ezek például az `oszt1`, `oszt2` és `oszt3`, akkor először az `funct.oszt1` metódust keresi az interpreter, ha nem találja akkor sorban a `funct.oszt2`, `funct.oszt3`, `funct.default` függvényeket próbálja megtalálni. Ha nem talált egyetlenegy megfelelő metódust se, akkor az hiba lesz. Implicit osztály esetén ugyanezt a keresési műveletet végzi el a `UseMethod` függvény. A függvényhívásnál a paraméterátadás során a paraméterek megfeleltetése csakis azok sorrendje alapján történik.

Az R jól dokumentált, így ha többet szeretnénk megtudni egy függvényről, akkor nagyon hasznos lehet az R súgórendszere. Használata nagyon egyszerű, a `help(amire.kivancsi.vagy)` függvény kiírja a keresett függvényről az információkat, amennyiben az a betöltött csomagok között van. Speciális karakterek esetén az argumentumot idézőjelek közé kell tenni. Előfordulhat, hogy egy függvény nevét nem tudjuk

pontosan, vagy a függvény nincs a betöltött könyvtárakban, ekkor jön jól a `help.search(keresett.fv)`, ez ugyanis az összes telepített csomagban keres.

3.2. Az nnet csomag

Neurális hálózatokat az R-ben többek között a `nnet` csomag segítségével hozhatunk létre. Először a csomagot be kell töltenünk a `library(nnet)` parancsal. A betöltés után használhatjuk a csomagban megtalálható főbb függvényeket, amelyek a következők: `nnet`, `nnetHess`, `multinom` és `predict`.

Új neurális háló objektumot az `nnet` segítségével tudunk létrehozni, ez a bemenő adatokhoz alkalmas, egy rejtett réteges előrecsatolt neurális hálót hoz létre. Két paraméterezéssel hívható:

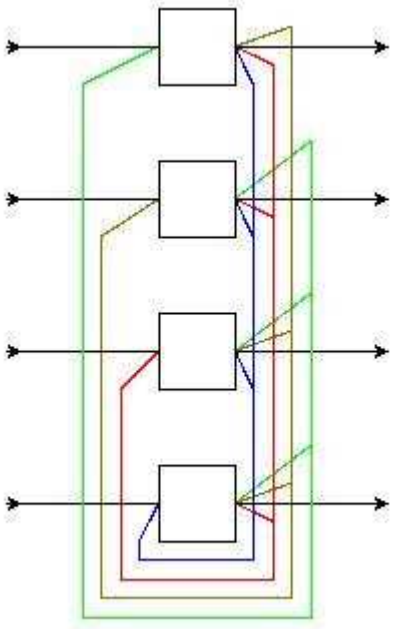
Az első az `nnet(formula, data, weights, ..., subset, na.action, contrasts = NULL)`, ebben az esetben a `formula`-ban `class ~ x1 + x2 +` … -ként definiált modell formula lesz a bemenő adat váza, ami alapján a függvény a `data`-ban megadott adatkeretből meghatározza a megfelelő bemenő adatokat. Ez azt jelenti, hogy bemenetként megadott adatkeretből az `nnet` függvény a `class` nevű oszlopot értelmezi majd az adatok osztályozásaként, míg az `x1, x2,` … nevű oszlopok lesznek a felhasznált adatok. A formulában a `class` tulajdonképpen az `x1, x2,` … adatok egy osztályozását jelenti. A `weights`-szel az egyes bemenő adatokhoz rendelhetünk súlyokat. A `subset`-el állíthatjuk be azokat az osztályokat, amik a tanításnál felhasználásra kerülnek. Amennyiben az adatok között előfordulhat `NA` érték is, akkor beállíthatunk egy ezt az esetet kezelő függvényt az `na.action` argumentum segítségével.

A másik paraméterezés az `nnet(x, y, weights, size, Wts, mask, linout = FALSE, entropy = FALSE, softmax = FALSE, censored = FALSE, skip = FALSE, rang = 0.7, decay = 0, maxit = 100, Hess = FALSE, trace = TRUE, MaxNWts = 1000, abstol = 1.0e-4, reltol = 1.0e-8, ...).` Itt `x` mátrix vagy adatkeret a bemenő adatokkal, `y` mátrix vagy adatkeret a várt kimenő adatokkal, `weights` mint az előző esetben az egyes példához rendelt súly, `size`-zal a rejtett rétegben lévő neuronok számát adhatjuk meg, `Wts` az inicializáló

vektor, ami ha hiányzik automatikusan véletlen számokból generálódik. A `mask`-kal azokat a paramétereket lehet megadni, amiket optimalizál a függvény. A `linout`, `entropy`, `softmax` és `censored` a tanítást befolyásoló logikai értékek, míg a `skip` az input és az output réteg között enged meg kapcsolatot. A `rang` argumentum megadása esetén a súlyok a [`-rang`, `rang`] tartományból veszik fel véletlenszerűen az inicializáló értékeiket. A `maxit` a tanítás során a maximális iterációs lépések száma. Ha a `Hess` igaz, a visszatéréskor kapott objektumban lesz egy `Hessian` attribútum, ami a végleges súlyokkal kiszámolt kétszeres deriváltakat tartalmazza, azaz a neurális háló Hesse-mátrixát. Ezt egyébként az `nnetHess()` függvénnyel is kiszámolhatjuk egy már létrehozott `nnet` objektumra. A `MaxNWts` a súlyok maximális számát állítja be, ezzel olyan hálózatok is létrehozhatók, melyek nagyok és számításigényesek. Végül az `abstol` és a `reltol` a tanításnál játszik szerepet, az iteráció nem folytatódik, ha az eltérés a várt kimenő adatoktól az `abstol` alá esik, vagy ha a `reltol` fölött van.

4. A Hopfield neurális háló és R implementációja

A John Hopfield által 1982-ben tervezett aszinkron háló n darab teljesen csatolt kétállapotú neuronból áll. Az n darab neuron egyetlen rétegben helyezkedik el, lényegében a bemeneti réteg egyben a rejtett és a kimeneti réteg is. A neuronok lehetnek binárisak, azaz $y_i \in \{0, 1\},\ i = 1, \ldots, m$, valamint lehetnek bipolárisak, azaz $y_i \in \{-1, 1\},\ i = 1, \ldots, m$. Én a reprezentációmban a bipoláris értékeket választottam. A hálózat neuronjai véletlenszerű sorrendben, de jól meghatározott algoritmus szerint változtatják az állapotukat -1 vagy 1 között, függetlenül a többi egységtől. Két neuron egyszerre sohasem vált állapotot. Ezzel ekvivalens működés érhető el, ha véletlenszerűen kiválasztunk egy egységet, majd kiszámoljuk a gerjesztettségét, és ennek megfelelően állítjuk be az állapotát. Azt, hogy a hálózat stabil állapotba kerüljön véges lépés után, a hálózat energiájának kiszámításával fogjuk bizonyítani. A hálózat teljesen csatolt, minden neuron önmagán kívül az összes többi neuronnal kapcsolatban áll. A súlyok reprezentálhatóak egy $n \times n$-es mátrixszal, ahol $W = \{w_{ij}\}$, a mátrix diagonálisa nulla, éppen azért, mert egy neuron sem áll kapcsolatban saját magával. A hálózat szimmetriájából adódik, hogy a w_{ij} súly megegyezik a w_{ji} súllyal.



4 neuronból álló Hopfield háló

4.1. A Hopfield háló energiája

Definíció. Legyen W az n neuronból álló Hopfield hálózat súlyainak mátrixa, θ a torzítások n elemű sorvektora. Ekkor a hálózat $E(x)$ energiája

$$E(x) = -\frac{1}{2} x W x^T + \theta x^T.$$

Az energia felírható még a

$$E(x) = -\frac{1}{2} \sum_{j=1}^{n} \sum_{i=1}^{n} w_{ij} x_i x_j + \sum_{i=1}^{n} \theta_i x_i$$

alakban is. A szorzás $1/2$-el a $w_{ij} = w_{ji}$ egyenlőségből adódik.

Tétel. Az n neuronból álló, aszinkron működésű Hopfield háló bármely induló kezdőállapotból mindig elér egy stabil állapotot az energiafüggvényének lokális minimumában.

Biz. Az $x = (x_1, x_2, \ldots, x_n)$ állapothoz tartozó $E(x)$ energiafüggvény adott. Ha a k-adik neuron lett kiválasztva az aktuális iterációs lépésben és ez az egység nem vált állapotot, akkor az energiafüggvény értéke nem változik, viszont ha állapotot vált, akkor az új $x' = (x_1, \ldots, x'_k, \ldots, x_n)$ megváltozott globális állapothoz tartozó energia $E(x')$-lesz. A két globális állapot energiájának különbsége

$$E(x) - E(x') = (-\sum_{j=1}^{n} w_{kj} x_k x_j + \theta_k x_k) - (-\sum_{j=1}^{n} w_{kj} x'_k x_j + \theta_k x'_k).$$

A belső szummában az $i = k$ eseten kívül megegyezik a két állapot, így ezen tagok különbsége is nulla, és mivel a $w_{kj} x_k x_j$ kétszer fordul elő a belső szummában, az $1/2$ is eltűnik a képlet elejéről. A $w_{kk} = 0$ miatt az egyenlőség felírható a

$$E(x) - E(x') = -(x_k - x'_k) \sum_{j=1}^{n} w_{kj} x_j + \theta_k (x_k - x'_k) = -(x_k - x'_k)(\sum_{j=1}^{n} w_{kj} x_j - \theta_k)$$

alakban. A $\sum_{j=1}^{n} w_{kj} x_j - \theta_k$ megegyezik a k-adik neuron ingerlési szintjével, így ha ezt elnevezzük e_k-nak (az angol excitation után) az előző képlet leegyszerűsödik a

$$E(x) - E(x') = -(x_k - x'_k) e_k$$

alakra. Az ingerlés e_k előjele különbözik x_k és $-x'_k$ előjelétől, mert ha nem így lenne, akkor az állapot nem változott volna, de az elején épp azt tettük föl, hogy megváltozott. Ebből következik, hogy mindkét esetben ha x_k negatív vagy pozitív, akkor

$$E(x) - E(x') > 0.$$

Ezzel bizonyítottuk azt, hogy ha egy neuron állapotot vált, akkor a hálózat energiája csökkenni fog. Mivel véges sok lehetséges állapot van, a hálózat mindig elér egy olyan állapotba, aminek energiáját nem lehet tovább csökkenteni.

4.2. A Hebb-i tanítás

A Hopfield hálót asszociatív memóriaként használhatjuk, és ha a hálózatunkat szeretnénk „beállítani” m darab stabil állapotra, akkor feladatunk a megfelelő súlyok megtalálása. Ennek a problémának a megoldására először a Hebb-i tanítási módszert ismertetem. A Hopfield háló álljon n neuronból, és a θ torzítás legyen nulla.

A Hebb-i tanítás során az alábbi formula alapján történik a súlyok változtatása. Az m darab n dimenziós kiválasztott $x_1, x_2, \ldots, x_m$ stabil állapotvektort betöltjük a hálózatba a következő szabály szerint:

$$w_{ij} \leftarrow w_{ij} + x_i^k x_j^k, \quad i, j = 1, \ldots, n, \quad i \neq j.$$

Az x_i^k és az x_j^k az x_k vektor i-edik és j-edik komponensét jelentik.

A tanítási módszert első lépésként csak az x_1 vektor felhasználásával mutatjuk meg, azaz csak egy stabil állapotot állítunk be a hálózatnak. Ebben az esetben a súlymátrixot a

$$W_1 = x_1^T x_1 - E$$

képlettel kapjuk, ahol E az $n \times n$-es egységmátrix. Mivel bármely bipoláris vektor esetén $x_i^k x_i^k = 1$, az egységmátrix kivonása garantálja, hogy a W mátrix diagonálisa nulla maradjon. Valamint az is nyilvánvaló, hogy a W_1 mátrix szimmetrikus. A W_1 súlyú Hopfield háló energiájának minimuma x_1-ben lesz, mert

$$E(x) = -\frac{1}{2} x W_1 x^T = -\frac{1}{2}(x(x_1^T x_1 - E)x^T) = -\frac{1}{2}(x x_1^T x_1 x^T - x x^T)$$

és a bipoláris vektorok esetén $x x^T = n$-ből következik, hogy az

$$E(x) = -\frac{1}{2} \parallel x x_1^T \parallel^2 + \frac{n}{2}$$

függvénynek lokális minimuma van a $x = x_1$-ben. Az egyszerűsítés után

$$E(x) = -\frac{n^2}{2} + \frac{n}{2}$$

marad, ami mutatja, hogy x_1-ben lokális minimum van.

Az m darab különböző $x_1, x_2, \ldots, x_m$ vektor esetén a W súlymátrixot a

$$W = (x_1^T x_1 - E) + (x_2^T x_2 - E) + \cdots + (x_m^T x_m - E)$$

képlettel kapjuk, amely egyszerűbben felírva

$$W = x_1^T x_1 + x_2^T x_2 + \cdots + x_m^T x_m - mE.$$

Ha az így betanított hálózatnak az x_1 vektort adjuk bemenő adatként, akkor az e vektor a neuronok izgatási szintje és

$$e = x_1 W = x_1 x_1^T x_1 + x_1 x_2^T x_2 + \cdots + x_1 x_m^T x_m - m x_1 E = (n-m)x_1 + \sum_{j=2}^{m} \alpha_{1j} x_j$$

Az $\alpha_{12}, \alpha_{13}, \ldots, \alpha_{1m}$ konstansok az első vektor skaláris szorzatát jelenti a maradék $m-1$ $x_2, x_3, \ldots, x_m$ vektorral. Az x_1 állapot stabil, amennyiben $m < n$ és a $\sum_{j=2}^{m} \alpha_{1j} x_j$ tényező kicsi. Ebben az esetben igaz, hogy

$$sgn(e) = sgn(x_1),$$

ahogyan azt feltételeztük. A módszert alkalmazhatjuk bármely másik vektorral. A legjobb eredményt a Hebb-i tanítással akkor lehet elérni, ha az $x_1, x_2, \ldots, x_m$ vektorok ortogonálisak, vagy közel vannak ahhoz.

4.3. A Hopfield háló implementációm

A Hopfield reprezentációmban a hálózat tulajdonságainak tárolására egy olyan listát használtam, melynek három elemet adtam meg és az osztályát a `"hopfield"` karaktervektorra állítottam. Tehát egy `hopfield` osztályú lista a következő három elemből fog állni: `input`, `weigths` és `theta`. Az `input` lesz az a mátrix vagy vektor, amivel a hálózat súlyait beállítom a Hebb-i tanítás szerint. A `weigths` a bemeneti vektorra beállított súlyok, míg `theta` az egyes neuronokhoz rendelt torzítások lesznek. A `theta` vektor a Hebb-i tanításban nem változik, definíció szerint nulla marad, viszont ha egy hálózatot kézzel szeretnénk beállítani, nem pedig tanítani, akkor tetszőleges torzítást meg lehet adni. Az elemeket soronként kell értelmezni, ez csak a mátrixoknál lényeges. Fontos hogy mindhárom elem numerikus értékű és ezeknek az elemei -1 és 1 értékek lehetnek.

Tehát a csomagomban új hopfield objektumot létrehozni a `hopfield( input.adatok)` függvénnyel lehet, ahol az `input.adatok` a beállítani kívánt állapotok mátrixa, vagy pedig egy vektor. A függvény a `weigthgeneral()` segédfüggvényt

használja a Hebb-i tanítás egy lépésének megvalósítására. Az R-ben írható rövid kódok kiváló példája ez a függvény, ami tulajdonképpen az $x^T x - E$ képletnek felel meg:

```
x %*% matrix( x, ncol=length(x), nrow=1) – diag(length(x));
```

A kivonás utáni részkifejezés az egységmátrixot állítja elő. A %*% jelen esetben a mátrixszorzás, melynek mindkét operandusának mátrixnak kell lennie, és mert az `x` egy vektor, ezt az R jelen esetben implicit 1 oszlopos mátrixszá konvertálja, míg az egysoros mátrixot nekünk kell explicit módon konvertálni.

Definiáltam továbbá egy `energy(hopfield,allapot)` generikus függvényt is, ami a kapott `hopfield` objektum inputként kapott állapotára kiszámolja az energiáját. Az `allapot`-nak vektornak kell lennie, szokásosan bipoláris értékekkel. A lényegi része ennek a függvénynek is rövid, két sor:

```
ret <- ( allapot %*% hopfield$weigths %*% allapot )*-0.5 +
hopfield$theta %*% allapot;
as.vector(ret);
```

Az első sor a $-\frac{1}{2} x W x^T + \theta x^T$ energia képletének felel meg, amelyet aztán vektorrá alakítva adok vissza.

Azt, hogy mit ad a hálózat egy input adatsorra a `predict(hopfield,adatok)` generikus metódus segítségével kaphatjuk meg. Az `adatok` lehet mátrix vagy vektor. A kód:

```
if ( is.vector(adatok) )
     adatok <- matrix( adatok, nr=1, nc=length(adatok) );
y <- matrix( stablestate.hopfield( hopfield, adatok[1,] ),
nr=1, nc=length(adatok[1,]));
i<-2;
for( i in 2:nrow(adatok) ){
     y <- rbind( y, matrix(stablestate.hopfield( hopfield,
adatok[i,] ), nr=1, nc=length(adatok[i,])) );
}
```

Az első résznél, ha vektort adtunk meg, akkor átalakítom egysoros mátrixszá, utána az első sorra meghívom a stabil-állapot kereső függvényt. Végül, ha a mátrixban több sor is van, a maradék sorra is meghívom az előbb említett függvényt. A mátrix i-edik sorának indexelése az `adatok[i,]`-vel, míg j-edik oszlopának hivatkozása `adatok[,j]`-vel történik. Továbbá az egyetlen eddig nem említett függvény az `rbind()`, ami a kapott argumentumait sorok szerint fűzi össze egy mátrixszá.

A stabil-állapot kereső függvényemben elsőként definiálok egy `lep` nevű változót, amiben az `adatvektor` hosszának megfelelő méretű mintát teszek, tehát ez egy n elemű vektor, amely

az 1,…,n számok egy permutációját tartalmazza. A ciklus addig fut, amíg a ciklusszámláló nagyobb nem lesz az `adatvektor` hosszától. Ez a ciklusváltozó újra egy lesz, valamint új mintát veszek, amennyiben a régi állapot energiájánál nagyobb annak az állapotnak az energiája, amelyet úgy kaptam, hogy a vektorom azon elemét az ellentettjére változtattam, amely a minta ciklusváltozó sorszámú elemével van indexelve. Ha nem nagyobb, akkor visszaállítom a régi állapotot, valamint növelem a ciklusváltozót. Így a ciklus befejeződik, ha nem változott meg egyik állapot se, ami azt jelenti, hogy stabil állapotban vagyunk. Ez pontosan a 4.1-es energiáról szóló fejezetben lévő bizonyítás leprogramozása, amikor is a hálózat neuronjai véletlenszerű sorrendben, de jól meghatározott algoritmus szerint változtatják az állapotukat -1 vagy 1 között, függetlenül a többi egységtől. Az előbb leírt `stablestate.hopfield(hopfield,adatvektor)` függvény:

```
lep <- sample(length(adatvektor));
lepesszam <- 1;
while( lepesszam <= length(adatvektor) ){
     regivektor <- adatvektor
     adatvektor[lep[lepesszam]] <-adatvektor[lep[lepesszam]]*-1
     if ( energy.hopfield( hopfield, regivektor ) >
energy.hopfield( hopfield, adatvektor ) ){
          lep <- sample(length(adatvektor));
          lepesszam <- 1;
     }
     else{
          adatvektor[lep[lepesszam]] <-
adatvektor[lep[lepesszam]]*-1;
          lepesszam <- lepesszam+1;
     }
}
adatvektor;
```

5. Csomagok készítése az R-ben

5.1. A csomagokról általában

A csomagok könyvtárakból töltődnek be. Ezeknek a könyvtáraknak installált csomagokat kell tartalmaznia. A könyvtárak alapértelmezett elérési útja az `R_install_könyvtar\library`. Más elérési utat mi is megadhatunk a `.libPaths(uj_eleresi_ut)` függvény segítségével.

A csomagok betöltésére a `library()` függvény szolgál, amelyet argumentum nélkül hívva a betölthető csomagok listáját kapjuk. Módunkban áll az alapértelmezettként betöltődő csomagok listájának módosítása is, méghozzá a `R_DEFAULT_PACKAGES` környezeti változóval, szintaktikailag egy vesszővel elválasztott lista megadásával.

Csomagot installálni a `install.packages(csomagnev)` segítségével lehet. Ha több elérési út van megadva a `.libPaths()`-nál, vagy nem az alapértelmezett könyvtárba szeretnénk telepíteni, akkor a függvénynek explicit kell megadni az installálási könyvtárat a `lib.loc` paraméter segítségével. Az előző függvényben megadhatjuk a `dependencies = TRUE` argumentumot is, amennyiben van olyan csomag, ami az installálandóhoz kell, viszont még nincs telepítve. Ekkor installáljuk a szükséges csomagokat is. Az `install.packages()` továbbá még `.tar.gz` fájlokból is tud telepíteni. Csomagot installálhatunk még a konzolból is az

```
R CMD INSTALL \ahova\installalsz csomag1 csomag2 ...
```

parancs segítségével, amely hasonlóan működik az előző függvényhez.

Csomagok és az R felépítéséhez Windowsban három eszköz szükséges:

- Pearl
- MinGW fordító

Ezek az eszközök mind megtalálhatóak a CRAN-on összegyűjtve, az `Rtools.exe`-ben. Valamint további eszközök kellenek a teljes, profi csomag készítéséhez, gondolok a PDF kézikönyvekre és HTML súgókhoz. Ezek lehetnek más fejlesztőeszközök is:

- Microsoft HTML Help Workshop
- LaTeX
- Inno Setup installáló

5.2. A csomagok szerkezete

A csomagnak tartalmaznia kell egy DESCRIPTION nevű fájlt illetve a következő alkönyvtárak közül majdnem mindet: R, data, demo, exec, inst, man, po, src, és tests. Ezen kívül még tartalmazhatja a következő fájlokat: INDEX, NAMESPACE, configure, cleanup, LICENSE, LICENCE, és COPYING. Az R figyelmen kívül hagyja a README, NEWS, és ChangeLog állományokat, ezek csak a felhasználóknak lesznek hasznosak.

A configure és cleanup script fájlok, amelyek az installálás előtt és után futnak le (ha a --clean opció meg volt adva).

A LICENSE, LICENCE, és COPYING fájlok opcionálisak és a licenceket tartalmazzák, például a GNU publikus licenc másolatát. Lehetőleg ne adjunk meg újabb saját licencet, helyette hivatkozzunk a share/licenses R-beli könyvtárban lévőkre, amelyeket a fejlesztők már leírtak.

A fájlok nevei legyenek helyesek az aktuális fájlrendszer szerint, illetve a csomag neve egyezzen meg a csomagot tartalmazó könyvtár nevével.

5.2.1. A DESCRIPTION fájl

A DESCRIPTION fájl alapvető információkat tárol a csomagról a következő formában:

Package: A csomag neve.

Version: A csomag verziószáma. Számjegyek elválasztva ponttal vagy kötőjellel.

Date: A verzió kiadási ideje yyyy-mm-dd formában.

Title: Rövid leírás a csomagról 65 karakterben, újsor és írásjelek nélkül.

Author: A csomag írója, leginkább humán olvasásra, és nem pedig automatikus feldolgozásra.

Maintainer: Egy név és egy e-mail cím a hibák jelentésére, azaz a csomag karbantartója.

Depends: Vesszővel elválasztott lista azokkal a csomagnevekkel, amelyektől ez a csomag függ. A nevet követheti a >= és <= relációs jel után megadott verziószám. Megadható csomagnévnek az R is, ekkor az R verziószámától függ a csomagunk. Egyéb függőségeket a SystemRequirements mezőben lehet megadni, vagy külön a README fájlban.

`Imports`: Azok a csomagnevek, amelyek nem szükségesek, de importálva lesz a névterük. Itt kell feltüntetni azokat a névtereket, melyekre a :: illetve a ::: operátorral a csomagunkban hivatkozunk, vagy a `Suggests` vagy az `Enhances` részben. Elméletileg az összes alapcsomagot ide lehetne írni.

`Suggests`: ugyan az a szintaxisa mint a `Depends`-nek, és azokat a csomagokat sorolja fel, amik nem feltétlen szükségesek, de esetleg a példákhoz vagy a címkékhez kellenek, esetleg egy függvény törzsében betöltött csomagok.

`Description`: Egymondatos leírás arról, mit csinál a csomag, és miért lehet hasznos.

`License`: licenc jogok.

`URL`: a csomag írójának honlapcíme, vagy a csomagról több információt tartalmazó honlap címe.

`Collate`: (vagy a platformfüggő verziója a `Collate.OStípus`, mint például a `Collate.windows`) segítségével szabályozhatjuk az R kódfájlok összeolvasási sorrendjét, amikor ezeket az R egy fájlba gyűjti össze a forráskódból való installálás során. Amennyiben ez a mező létezik, akkor az összes R forrás fájlt fel kell sorolnunk benne, ami a csomagunkban van.

`ZipData`: Ez a mező az automatikus Windows-os csomagépítésnél az adatkönyvtár becsomagolását vezérli. Állítsuk `"no"`-ra ha a csomagunk nem működik tömörített adatkönyvtárral.

```
Package: hopfield
Version: 1.0
Date: 2008-04-20
Title: Hopfield halok csomag
Author: Kovacs Tibor Krisztian
Maintainer: Kovacs Tibor Krisztian <tibork1@t-online.hu>
Description: A csomag Hopfield halok letrehozasara keszult
License: GPL-3
```

`DESCRIPTION` fájl tartalma a csomagomban

Az előzőekben leírt mezők közül a `Package`, `Version`, `License`, `Description`, `Title`, `Author` és a `Maintainer` kötelező, a többi opcionális. A maximális hordozhatóság elérése miatt ezt a fájlt ASCII karakterkódolással írjuk meg. Amennyiben nem csak ASCII karaktereket használtunk, akkor a `DESCRIPTION` fájlunknak tartalmaznia kell egy `Encoding` mezőt, amely nem csak a `DESCRIPTION` fájl saját kódolását

adja meg, hanem az érvényes lesz a csomagban használt fájlok kódolásánál is. A következő kódolásokról tudják a fejlesztők, hogy a hordozhatóság nem sérül: Latin-1, Latin-2 és UTF-8.

5.2.2. Az Index fájl

Ez az opcionális fájl tartalmaz egy-egy sort minden kellőképpen érdekes objektumról a csomagban, soronként megadva az objektum nevét és egy leírást róla (többek között az olyan kiíratásra szolgáló függvényeket, melyek ritkán vannak explicit meghívva, nem lényeges feltüntetni ebben a fájlban). Normális esetben ez a fájl hiányzik, és az ennek megfelelő információk automatikusan legenerálódnak a dokumentáció forrásfájljaiból, ha az `Rdindex()` függvényt használjuk a csomageszközök közül, mikor forrásfájlból installálunk, vagy a csomagépítőt használjuk.

Ezt a fájlt létrehozni leginkább akkor érdemes, amikor egy egyedi igényekre szabott áttekintő súgó fájlt szeretnénk létrehozni a csomagunkról.

5.2.3. A csomagok alkönyvtárai

Az `R` csomagalkönyvtár csakis R forráskód fájlokat tartalmaz. A telepítendő kódfájlok neveinek ASCII betűvel vagy számjeggyel kell kezdődnie, és `.R`, `.S`, `.q`, `.r` vagy `.s`-re kell végződnie. A fejlesztők a `.R`-et ajánlják, mert azt elvileg semmilyen más ismert program nem használja. A csomagalkönyvtárakban található fájlok neve és az általuk létrehozott R objektumok nevei között nincs összefüggés, így azok nyugodtan eltérhetnek egymástól. Lehetőleg a csomagunkban lévő R kódfájlok csak maguk hozzanak létre R objektumokat, és semmiképpen se hívjanak olyan függvényeket, melyeknek mellékhatásai lehetnek. Amennyiben számításokhoz szükséges objektumokat szeretnénk létrehozni, akkor ezek a kódban „korábban” legyenek definiálva (lásd a `Collate` mező leírásánál az 5.2.1 es fejezetben).

A kódfájlokon kívül két másik típusú fájl lehet ebben az alkönyvtárban. Egyik a `sysdata.rda`, ez a fájl olyan adatokat tartalmaz, amelyekre a csomagunknak a működéshez van szüksége, és amelyeket nem szeretnénk elérhetővé tenni felhasználóknak a

`data` alkönyvtáron keresztül. Valamint az `.in`-re végződő fájlok is megengedettek ebben a könyvtárban, hogy a később leírt `configure` script segítségével megfelelő fájlok generálódhassanak le.

A kódfájlokban csakis ASCII karaktereket és whitespace karaktereket használjunk. Egyéb karaktert csak a megjegyzésbe írjunk, ekkor viszont előfordulhat, hogy az a megjegyzés nem lesz olvasható például UTF-8 környezetben. A nem ASCII karakterek az objektumok neveiben hibához vezetnek a csomag installálása során. Valamint a \ jel után bármilyen bájt megengedett, de mellőzzük a `\uxxxx` escape karakterek használatát.

Különféle R eszközöket használhatunk a csomagban inicializálásra és feltakarításra (clean up). Névterek nélküli csomagok számára ezek a `.First.lib()` és `.Last.lib()` függvények. Szokásos módon ezek a függvények a `zzz.R` nevű fájlban vannak definiálva. Amennyiben a `.First.lib()` definiálva van, akkor a csomagunk könyvtárának nevével és a csomagunk nevével, mint argumentumokkal hívódik meg, miután a csomag betöltődött és az objektumok csatlakoztatva (attach) lettek. A csomag csatlakoztatása alatt a csomag névterének hozzáadását kell érteni ahhoz a keresési útvonalhoz, amiben az R az objektumok nevét keresi. A névtér a már betöltött csomagok elé, de a felhasználói munkaterület mögé lesz beszúrva. Amennyiben a csomag verziószámmal települt, akkor a csomagnév argumentum tartalmazza a verziószámot is, például `"ash_1.0.9"`. A megszokott használata a `.First.lib()`-nek, hogy belsejében meghívjuk a `library.dinam()` függvényt (más nyelven írt) lefordított kódok betöltésének érdekében, vagy olyan függvények hívására, amelyeknek mellékhatásaik is vannak. A `.Last.lib()` ha létezik a csomagban, pontosan a csomag leválasztása (detach) előtt hívódik meg, argumentumként kapva a csomag installálási helyének a teljes elérési útját. Leválasztás alatt a csomag névterének eltávolítását értem a keresési útvonalról. Ritka dolog a csomagok leválasztása, és az is hogy a csomagban van `.Last.lib()` függvény. Egy használati módja ennek a függvénynek, hogy `library.dynam.unload()`-ot hívjuk a belsejében a lefordított kódok kitöltésére.

A `man` alkönyvtár csakis R dokumentációs (Rd) fájlokat tartalmaz a csomagban lévő objektumokhoz. A dokumentációs fájlok neveinek ASCII betűvel vagy számjeggyel kell kezdődnie, és `.Rd` vagy `.rd`-re kell végződnie, továbbá érvényesnek kell lennie ’file://’ URL címekben, ami azt jelenti, hogy teljesen ASCII kódolásúnak kell lennie és nem tartalmazhat `%` jelet. Fontos hogy az összes felhasználó által elérhető objektum dokumentálva legyen. Ha egy `csom` csomag tartalmaz olyan felhasználó által elérhető objektumokat, amelyek csak „belső”

használatra valóak, akkor ebben a könyvtárban lennie kell egy `csom-internal.Rd` fájlnak, amiben ezek az objektumok le vannak írva, feltüntetve azt is, hogy ezeket ne hívja a felhasználó. Az ilyen belső használtra való objektumokat a csomagokban rejtsük el egy névtérben, így dokumentálni sem kell őket.

Mind az `R`, mind a `man` könyvtárban lehetnek OS függő alkönyvtárak, mint például a `windows` vagy `unix`.

A lefordított (azaz más nyelven megírt) kódokhoz tartozó forrás és header fájlok helye az `scr` alkönyvtárban van (a header fájloknak a fejlesztők mind C++, mind Fortran 9x esetén a `.h` fájlvégződést ajánlják), ide rakhatók még opcionálisan a `makevars` és `makefile` állományok. Amikor a csomagot `R CMD INSTALL` paranccsal telepítjük, ezek a `make` fájlok arra szolgálnak, hogy a fordítást és a linkelést vezéreljék az R-be való betöltés során. A betöltési folyamat elősegítésére vannak definiált alapváltozók és szabályok, amik az R installálása során állítódtak be, és az `R_HOME\etc\Makeconf` fájlban találhatóak. Ezek a változók segítséget nyújtanak a C, C++, FORTRAN 77, Fortran 9x kódok betöltéséhez és használatához. Az R telepítése során beállított szabályokon a `Makevars` fájlunkban változtathatunk makrók definiálásával. Az előző fájl elég általános eszközöket biztosít ahhoz, hogy ne kelljen használnunk egy csomagspecifikus `Makefile`-t. Amennyiben mégis készítettünk ilyen `Makefile`-t, akkor annak annyira általánosnak kell lennie, hogy működjön az összes R platformon. A platformspecifikus fájlnevek használata esetén az aktuális platformon nagyobb precedenciája lesz az ilyen fájlnak, például a `Makevars.win` lesz érvényes Windowson a sima `Makevars` fájl helyett.

A `data` alkönyvtárban vannak a csomag által szolgáltatott plusz adatfájlok, amelyeket a `data()` függvénnyel tölthetünk be az R-en belül. Az adatfájlok típusa a következő három közül lehet az egyik: R kód `.R` vagy `.r` kiterjesztéssel, táblázatos formájú adatokat tároló állományok `.tab`, `.cvs`, `.txt` végződéssel vagy a `save()` függvénnyel lementett objektumok `.RData` vagy `.rda` végződéssel. Az összes R verzió ugyanolyan bináris tömörítési eljárást használ (az XDR-t), így bármelyik verzió be tud olvasni tömörítetten kimentett objektummásolatokat. Lényeges, hogy ezeknek az adatfájloknak ugyanúgy használhatónak kell maradnia akkor is, ha a tartalmazó csomagot nem is töltöttük be. Az adatkönyvtárban lévő `00Index` állománynak az összes ebben a könyvtárban lévő, adatokat tartalmazó objektumok nevét tartalmaznia kell, mindegyikhez egy rövid leírást mellékelve.

Az újabb verziókban nem kell `00Index` fájlt írnunk az adatalkönyvtárba, mert az automatikusan generálódik a dokumentációs forrásfájlokból, mikor a forrásból installálunk, vagy az `R CMD build` parancsot használjuk. Amennyiben az adatfájlok nagyon nagyok, akkor meggyorsíthatjuk az installálást a `datalist` fájl megadásával.

A `demo` alkönyvtár R scripteket tartalmaz, amelyek a `demo()` függvénnyel futtathatók az R-ben, és a csomag valamilyen funkcionalitását mutatják be. Ezek a demók interaktívak is lehetnek, így nincsenek automatikusan leellenőrizve, ellenben a `tests` könyvtárban lévő, direkt tesztelésre szánt kódokkal. Az itt található scripteknek R szintaktikának megfelelőnek kell lennie, és a fájlok neveinek `.R` vagy `.r`-re kell végződnie. Amennyiben van `demo` könyvtárunk, írnunk kell hozzá egy `00Index` fájlt is, ami soronként tartalmazza a demók neveit és a hozzá tartozó rövid leírást. Figyeljünk oda, mert ezt a fájlt nem generáltathatjuk automatikusan.

Az `inst` könyvtár tartalma rekurzívan átmásolódik az installálás helyére. Ez a másolás az `src` felépítése után történik, szóval az ott lévő `Makefile` még létrehozhat installálandó fájlokat. A következő fájlok viszont nem lesznek installálva: `INDEX`, `LICENSE`/`LICENCE` és `COPYING`, így a Windows és a MacOS felhasználók nem találkoznak vele a lefordított csomagban, valamint azok sem, akik az `R CMD INSTALL` parancsot vagy az `install.packages()` függvényt használják. Tehát minden olyan fájlt, amit szeretnénk hogy a felhasználó lásson, azt ebbe a könyvtárba kell rakni.

A `tests` könyvtárban találhatóak a csomagspecifikus tesztelő kódok. Ezek a teszt kódok lehetnek `.R` vagy `.Rin` fájlokban is, és lényegében olyan függvényhívások a csomag összes objektumára, amelyeket szélsőséges paraméterezéssel látunk el. Az eredményt `.Rout` végződésű fájlban kapjuk meg. Amennyiben van egy `.Rout.save` fájl a könyvtárban, akkor az előző eredmény összehasonlítódik ezzel, majd az eltérések naplózva lesznek, de nem okoznak hibát. A tesztek a csomag installálása után lefutnak és beállítódik az `R_LIBS` környezeti változó.

Továbbá van még két alkönyvtár, a `po` és az `exec`. Az `exec` tartalmazhat a csomagot kiegészítő egyéb futtatható állományt, tipikusan script fájlokat az interpretereknek, mint a Perl vagy a Tcl. Ezt egyébként nagyon kevés csomag használja, és ráadásul csak kísérleti fázisban van. A `po`-nak a csomagok nemzetköziesítésében (I18N) van szerepe.

5.3. Konfiguráció és cleanup

Amennyiben a csomagnak szüksége van néhány rendszer-specifikus beállításra a feltelepítés előtt, akkor írnunk kell egy `configure` (Bourne shell) script fájlt, amelyet majd az `R CMD INSTALL` fog végrehajtani azelőtt, hogy bármi is történne az installálás során. Ez akár lehet egy olyan script fájl is, amit az Autoconf készített, ami egy Bourne shell scriptet létrehozó program forrásfájlok konfigurációjához, de lehet egy a felhasználó által írt script is. Ezt arra használhatjuk, hogy észleljük a nem standard könyvtárak létezését, hogy a megfelelő kódokat installálásnál kikapcsolhassuk a csomagban, hogy azok ne küldjenek hibaüzeneteket a csomag felépítése vagy használata közben. Ha valaki nem akarja megírni ezt a script fájlt, akkor az Autoconf teljes mértékben a rendelkezésre áll a csomagok kiegészítéséhez (például változók cserélése, könyvtárak keresése stb.).

A másik (Bourne shell) script fájl a `cleanup`. Ezt is az `R CMD INSTALL` parancs futtatja a telepítés legvégén, amennyiben jelen van ez a fájl és a `--clean` paraméter is meg volt adva. Ezt a fájlt egyébként az `R CMD build` is végrehajtja, amikor forráskódból építjük fel a csomagunkat. Arra használható, hogy feltakarítsuk a forrásfájlok könyvtárait. A `configure` által létrehozott fájlokat mind törli.

Lássunk egy példát a `configure` használatára! Tegyük fel, hogy egy olyan `pelda` nevű alkönyvtárt szeretnénk használni az `src`-n belül, ami C forráskódokat tartalmaz. Az Autoconf programmal készíthetünk egy olyan script fájlt, amely megkeresi a `pelda` könyvtárat, és annak megfelelően, hogy megtalálta-e vagy sem, a `van_pelda_konyvtar` valtozót beállítja `TRUE`-ra vagy `FALSE`-ra. Majd ezt a `@van_pelda_konyvtar@` változót megkeresi az `.R.in` forráskódokban, és a behelyettesített változó érétkével létrehozza az `.R` fájlokat. Ekkor, ha a `pelda.R.in` fájl a következő kódsorokat tartalmazza:

```
pelda <- function(x){
     if (!@van_pelda_konyvtar@)
          stop(”Nincs pelda konyvtar”)
     …
},
```

akkor a `configure` fájlsegítségével az `R CMD INSTALL`, ha nem találhato a `pelda` könyvtár, a következő kódsorokat tartalmazó `pelda.R` fájlt hozza létre:

```
pelda <- function(x){
     if (!FALSE)
          stop(”Nincs pelda konyvtar”)
     …
}
```

Így ha nincs `pelda` könyvtár, akkor a `pelda()` függvény a `”Nincs pelda konyvtar”` üzenetet fogja kiírni.

Előfordulhat néha, hogy nem mindig jól fog működni a konfigurációs script fájl Windows rendszeren, legalábbis az Autoconf által készített scriptekről ez általánosan elmondható, de a normális shell scriptek működnek. Amennyiben a csomagunkat nyilvánosan elérhetővé szeretnénk tenni, akkor tájékoztassuk a nem Unixos felhasználókat a konfigurálásról, vagy pedig készítsünk egy `configure.win` fájlt. Opcionálisan csinálhatunk akkor egy `cleanup.win` fájlt is.

5.3.1. `Makevars` használata

Néha a saját `configure` fájl megírása elkerülhető a `Makevars` fájl megírásával, ugyanis a `configure` fájl egy másik tipikus felhasználása a `Makevars` létrehozása a `Makevars.in` fájlból.

A `Makevars` legszokásosabb használata, hogy beállítsunk vele előfeldolgozó flag-eket a `PKG_CPPFLAGS`-en keresztül, beleértve elérési utakat is. Valamint C, C++, FORTRAN fordítóknak megadható flag-eket is sorban a `PKG_CFLAGS`, `PKG_CXXFLAGS` és `PKG_FFLAGS` változókon keresztül. Valamint a linkernek is át tud adni flag-eket a közös objektumok felépítésekor, például az `–L` és `–l` opciókat a `PKG_LIBS` segítségével. Ha egy olyan csomaghoz írunk `Makevars` fájlt, amelyet aztán közkézre akarunk adni, figyeljünk arra, hogy ne legyen a mi fordítónkra specifikálva, azaz más verziójú fordítóval is működjön. A következő FORTRAN makrókat használhatjuk a `Makevars` fájlban: `FLIBS`, `BLAS_LIBS`, `LAPACK_LIBS`, `SAFE_FFLAGS`. De mivel nem ismerem túlságosan a FORTRAN rendszert, ezért nem is tudom bővebben kifejteni az előző makrókat.

A fent leírt flag-eket legtöbbször a változónév után egyenlőségjellel adjuk meg, majd azt írjuk be, pontosan úgy, ahogy a parancssorban írnánk az adott nyelv fordítójának.

5.4. A csomag ellenőrzése

A csomag ellenőrzésére az `R CMD check` parancs való (ez az R csomagellenőrzőt indítja el), ez lényegében ugyanazt teszi, mint ha felinstallálnánk a csomagot, csak sokkal részletesebb hibaüzeneteket ad. Tehát az R csomagellenőrzője teszteli, hogy a forráskód megfelelően működik-e. Futtatható egy vagy több könyvtáron, vagy gzip-pel becsomagolt `.tar.gz` vagy `.tgz` fájlokon. A következő ellenőrzéseket végzi el:

1. Megnézi hogy a csomag installált-e. Figyelmeztet a hiányzó kereszthivatkozásokra és a duplikált objektumleírásokra a súgó fájlokban.

2. Ellenőrzi hogy a fájlnevek helyesek-e a támogatott operációs rendszerekben.

3. A fájlok és a könyvtárak rendelkeznek-e a megfelelő jogokkal? (Csak Unix alatt)

4. Ellenőrzi a `DESCRIPTION` fájl teljességét, és néhány mező helyességét is. Hacsak nem hagyjuk ki az installációs tesztet, az ellenőrzés leállítódik, ha a csomagfüggőségeket nem lehet futási időben megoldani. Ellenőrzi hogy a csomag neve nem-e egy standard könyvtár neve, és nem-e egy a forgalomból kivont standard könyvtárak nevei közül (ezek a `ctest`, `eda`, `lqs`, `mle`, `modreg`, `mva`, `nls`, `stepfun` és `ts` könyvtárnevek), amelyeket a `library()` függvény speciálisan kezel. A másik ellenőrzés, hogy a `Depends`, `Imports`, `Suggests` vagy `Contains` mezőkben a csomag működéséhez szükséges összes csomag fel van-e sorolva.

5. A címkékben és a demókban elérhető index információkat ellenőrzi a teljességre.

6. A könyvtárakat ellenőrzi, hogy megfelelőek-e a fájlnevek valamint nem-e üresek. A fájlneveken végzett ellenőrzés a `--check-subdirs=ertek` opció beállítása szerint működik. Alapból ez az érték a '`default`', de beállíthatjuk '`yes`'-re és '`no`'-ra is. Az `src` könyvtárra nem fut le az ellenőrzés, amennyiben van `configure` script fájlt és '`yes-maybe`' értéket adtunk meg. Nem fut le akkor sem az `src`-n, ha van `Makefile` vagy `Makefile.in`. Az `R` könyvtárban megengedettek a `.in` fájlok.

7. Az R fájlokat ellenőrzi szintaktikailag. Azok a bájtok, amelyek nem ASCII karakterek, csak figyelmeztetést okoznak, de vegyük őket hibának, ha nem tudjuk, hogy a csomag mindig ugyanabban a környezetben fog futni.

8. Ellenőrzésre kerül, hogy a csomag be tud-e töltődni, először az alapbeállításban betöltődő csomagokkal, majd utána csak a `base` alapcsomaggal. Ha a csomagnak van

névtere, ellenőrződik, hogy ez be tud-e töltődni egy olyan üres munkaszakaszba, ahova csak a `base` csomag névtere van betöltve.

9. Az R fájlokat leellenőrzi, hogy helyesen hívták-e meg a dinamikus könyvtárakban lévő lefordított kódokat a `library.dynam()` segítségével. Az összes külső függvényhívás (`.C`, `.Fortran`, `.Call` és `.External` függvények esetén) tesztelődik, hogy van-e PACKAGE argumentuma, és ha nincs, vajon a Windowsban a megfelelő DLL fájl a csomag névteréhez tartozik-e. Minden más hívás feljegyződik. (A fejlesztők figyelmeztetnek, hogy mindig használjuk a `library.dynam()` PACKAGE argumentumát.)

10. Ellenőrzi az Rd fájlok helyes szintaktikáját, a meta-adatokat, és a kötelező mezők jelenlétét (`\name`, `\alias`, `\title`, `\description` és `\keyword`). A `\name` és `\title` üres-e, a `\keyword` kulcsszavakat összehasonlítja az általános kulcsszavakkal (amik a `R_home\doc\`KEYWORDS fájlban találhatóak), és keres hiányzó kereszthivatkozásokat is.

11. Ellenőrződik, hogy minden felhasználó által látható objektumhoz tartozik-e bejegyzés a dokumentációs fájlokban.

12. A függvények, adathalmazok és S4 osztályok (ezek valósítják meg a tényleges objektumorientáltságot az R-ben) dokumentációit leellenőrzi szintaktikailag, hogy a hozzájuk tartozó kóddal konzisztensek-e.

13. Az összes, a dokumentáció `\usage` részében megadott függvény argumentuma összevetésre kerül a dokumentáció `\arguments` részében lévőkkel, és hibát eredményez az eltérés.

14. Az összes C, C++ és FORTRAN header és forrás fájl sorvége karakterei leellenőrződnek (csak LF lehet). Amennyiben az `src` könyvtárban vannak `Makefile` és `Makevars` fájlok és ezek `.in`-re végződő változatai, akkor azok sorvége karakterei is ellenőrzésre kerülnek, valamint a bennük lévő '`$(BLAS_LIBS)`' változó helyes használata is.

15. A dokumentáció `\examples` részében lévő példakódokat is leteszteli, hogy futnak-e. Természetesen a kiadott csomagnak képesnek kell lennie saját példakódjainak futtatására. Minden példakódot tiszta környezetben, azaz új munkafelületen futtatja az ellenőrző.

16. Ha a csomag forráskódjai tartalmaznak `tests` könyvtárat, akkor a könyvtárban lévő forrásfájloknak is futniuk kell.

17. A csomag címkéiben lévő kódrészleteket is ellenőrzi.

18. Ha elérhető LaTeX a gépen, akkor a súgó fájlokból az `R CMD check` felépítit a `.dvi` fájlokat ellenőrzésképpen.

További segítséget kaphatunk a `check`-ről és az általa végzett folyamatokról az `R CMD check --help` parancs segítségével. Valamint azt is tudnunk kell, hogy flag-ekkel beállíthatjuk az ellenőrizendő lépéseket.

5.5. A csomag felépítése

Az R forrásfájlokból felépíteni a csomagot az `R CMD build` parancs segítségével lehet. Inkább a valójában megszokott .tar.gz fájlból építsük fel a csomagot. Néhány ellenőrző és feltakarító lépést végrehajt a `build`. Például az ellenőrződik, hogy az objektum-hivatkozások léteznek-e. A C, C++ és FORTRAN forrásfájlok és a hozzájuk tartozó `make` fájlok letesztelődnek, és a sorvége karakterek bennük átkonvertálódnak LF karakterré, ha szükséges. A futásidejű csomagtesztelést inkább a `check`-kel végezzük.

Hogy kizárjunk fájlokat a csomagunkból felépítés közben, írhatunk egy listát a kizárandó mintákkal az `.Rbuildignore` fájlba, a csomag főkönyvtárába. Ezek a minták olyan Perl reguláris kifejezések, amelyekre illeszkedő nevű fájlokat kizár az `build`. A fájlnak minden sorban egy mintát kell tartalmaznia. Továbbá a `CVS`, `.svn`, `.arch-ids`, valamint a `GNUMakefile` fájl automatikusan ki lesz zárva, valamint a `.`#-val kezdődő, vagy #-val kezdődő vagy végződő, vagy ~-ra, `.bak`-ra, `.svp`-re végződő nevű fájlokat is alapból kizárja a `build`. Azokat a fájlokat is kizárja a `build`, amiket a `check` jelölt meg hibás karakterek miatt flag-ekkel, és az `R`, `demo` és `man` könyvtárakban vannak. További ellenőrzést végez a `build` üres csomagkönyvtárak után. Ha ilyet talál, akkor azt töröljük le és futtassuk újra a `build`-et. Az előző eset elkerülésére lehet hasznos egy végső ellenőrzés futtatása a csomag tartalmára az `R CMD check --check-subdirs=yes` parancsal.

Az `R CMD build` szintén alkalmas előfordított bináris fájlokból álló csomag felépítésére, de részesítsük előnyben az `R CMD INSTALL --build` parancsot, mert sokkal rugalmasabb. Sőt, a Windowsos felhasználóknak inkább az utóbbi használata ajánlott.

5.6. A csomagépítés és ellenőrzés beállításai

A meglévő konzolos opciókon túl az `R CMD check` beállításai tovább finomíthatók (Pearl) változók beállításával egy konfigurációs fájlban. Ezt megadhatjuk a `--rcfile` opció után, vagy használjuk az alapértelmezett `$HOME\.R\check.conf` fájlt (ezeket a változókat én az `R_HOME\bin\check` fájlban találtam meg), de ez azt feltételezi, hogy a `HOME` változó be van állítva. Tehát a fájlban a következő konfigurációs változók érhetőek el:

`$R_check_use_install_log`: Ha igaz, akkor a kimenet eltárolódik egy log fájlban, ami alapbeállításban a `00install.out`. Alapbeállítása `TRUE`.

`$R_check_all_non_ISO_C`: Ha igaz, akkor a nem ANSI C fájlok miatti fordítói figyelmeztetések nem lesznek figyelmen kívül hagyva. Alapbeállítása `FALSE`.

`$R_check_weave_vignettes`: Ha igaz, akkor a csomagcímkéket összeolvassa az ellenőrző feldolgozáskor. Alapbeállítása `TRUE`.

`$R_check_latex_vignettes`: Ha igaz, akkor a LaTeX-ben írt csomagcímkék is ellenőrzésre kerülnek. Alapbeállítása `TRUE`.

`$R_check_subdirs_nocase`: Ha igaz, akkor a csomagunk könyvtárainak neveiben lévő kis és nagybetűk is összehasonlításra kerülnek a referenciában megadott könyvtárnevekkel. Alapbeállítása `FALSE`.

`$R_check_subdirs_strict`: A `--check-subdirs` opció inicializáló értékét lehet megadni. Alapbeállítása `default`.

`$R_check_force_suggests`: Ha igaz, akkor a csomag működéséhez szükséges más csomagok hiányzása esetén hibát jelez. Alapbeállítása `TRUE`.

`$R_check_use_codetools`: Ha igaz, akkor a `codetools` csomagot is használni fogja a `check` az objektumok láthatóságának részletes ellenőrzésére. Alapbeállítása `TRUE`.

`$R_check_Rd_style`: Ha igaz, akkor az Rd fájlok S3 metódusokra való hivatkozását ellenőrzi. Alapbeállítása `TRUE`.

`$R_check_Rd_xrefs`: Ha igaz, akkor az `.Rd` fájlok kereszthivatkozásai ellenőrzésre kerülnek. Alapbeállítása `TRUE`.

A későbbi R verziókban az `R CMD build` parancshoz is lehet használni egy hasonló sémát a telepítés finomhangolására. Valamint vannak további belső beállítások, amiket az

_R_CHECK_*_ környezeti változókon keresztül érhetünk el, de ahhoz a Perl forrásfájlokat kell jobban ismernünk.

5.7. Csomag névterek

A csomag névterek három dologra jók. Az első, hogy a csomag írója elrejtheti azon függvényeit, amelyeket nem szeretne, hogy a felhasználó elérjen. A második, hogy elkerülhetünk velük olyan hibákat, amik akkor fordulnak elő, amikor a felhasználó véletlenül pont egy olyan nevű függvényt definiál, amit a csomagunk már definiált (névütközés). Az ilyen esetben a csomagunk függvényét elérni a `csomagnev::fvnev` alakban tudjuk az R munkafelületről. Végül a csomagunkon belül megkönnyíti az objektumainkra való hivatkozást, ugyanis a csomagon belül a saját objektumainkat nem kell a `::` operátorral hivatkozni. Az R a keresési útvonalat mindig a névterek alapján építi fel, először mindig a felhasználói munkaterületben keres, majd sorban a többi betöltött csomagban, végül a `base` alapcsomagban.

Az R-nek van egy névtér kezelő rendszere csomagok számára. Ez azt jelenti, hogy a csomag fejlesztői eldönthetik, mely változókat exportálják, azaz teszik láthatóvá a felhasználók számára, és melyeket importálják más csomagokból. Jelenleg a csomag számára névteret úgy hozhatunk létre, hogy csinálunk egy `NAMESPACE` fájlt a csomag főkönyvtárában. Ez a fájl névtér direktívákat tartalmaz, amik leírják a csomag névterébe importált és onnan exportált objektumok neveit. Továbbá direktívákat a közös objektumok regisztrálásához, és az összes S3 típusú metódushoz, ami a csomagunkban található. A fájl tartalmilag hasonlít egy R script fájlra, de nem R kódként lesz feldolgozva, ezért benne legfeljebb csak a nagyon egyszerű if szerkezet használható.

A névtérrel rendelkező csomagokat is hozzáveszi a keresési útvonalhoz a `library()` függvény. Csak az exportálandó változók kapnak helyet egy külön keretben. Viszont ha olyan csomagot töltünk be, amiben egy másik csomagból importálunk változókat, akkor az a csomag is betöltődik (ha még nem volt betöltve), de ez viszont nem fog hozzáadódni a keresési útvonalhoz.

Ha a névtereket egyszer már betöltöttük, akkor azok lezáródnak, ami azt jelenti, hogy nem lehet a továbbiakban többet exportálni vagy importálni, valamint a belső változók kötése

nem változtatható. Ez a lezárás lehetővé teszi a névtér mechanizmus egyszerű implementációját, továbbá a kódelemző és fordító eszközök számára egy függvénytestben hívott globális változó megfelelőségének könnyű azonosítását.

A csomaghoz adott névtér változtat a csomagon belül használt változók keresési stratégiáján. A keresés a csomag névterében indul, folytatódik az importálásokban, majd a főcsomag névterében, végül a további normál útvonal jön.

A NAMESPACE fájlban exportálásra az `export(valtozok)` parancsot használhatjuk, a benne vesszővel elválasztottan felsorolt változók lesznek exportálva. Amennyiben sok változót akarunk exportálni, használhatjuk a `exportPattern("regexp")` parancsot, ami a `regexp` reguláris kifejezésben megadott mintára illeszkedő összes változót exportálja. A névtérrel rendelkező csomag implicit importálja a `base` csomag névterét. Azokat a változókat, amelyek más csomagban exportálva vannak, explicit kell importálnunk, amit az `import(valtozo)` segítségével tehetünk. Másik fajtája az `importFrom(csomagnev, valtozo)` ami a megadott `csomagnev` csomagból importálja a változót. Lehetséges olyan változót exportálni, amit másik csomag importál be.

Amennyiben van olyan változó, amelyet nagyon ritkán használunk a kódunkban, akkor van lehetőség az importálás elhagyására, helyette a kódban minősítve kell hívni azt, a `csomagnev:::valtozo` formában. Bár ez kevésbé hatékony és elvész az az előny is, hogy az összes függőség egy helyen, a NAMESPACE fájlban legyen, szóval ez a megközelítés nem ajánlott. Előnye az előző hivatkozásnak, hogy késleltethetjük a nagyon ritkán használt csomag betöltődését.

Az S3 metódusoknak a standard függvénye az egy `UseMethod()`-ból álló függvénytörzsű függvény. Az e függvény által végzett illesztés esetleg sikertelen lehet, az olyan importált csomagok esetén, amelyek nincsenek hozzáadva a keresési úthoz. Hogy biztosak legyünk abban, hogy a `UseMethod()` megtalálja ezeket a metódusokat a keresési útvonalon, regisztrálnunk kell az ilyen függvényeket a NAMESPACE fájlba az `S3method(generikusfvnev,csomagnev)` segítségével. Ezt a mechanizmust direkt a generikus függvények használatához szánták a készítők a névterekben.

Számos horogfajta (hooks) van, amely támogatja a névtereket. A horgok arra valók, hogy segítségükkel a közös objektumokat töltsünk be az R-be. Pár fejezettel előbb már írtam, hogy névtérrel rendelkező csomagokban ne használjuk a `.First.lib()` függvényt, mivel a betöltés és a csatlakoztatás két különböző művelet, ha névtereket használunk, mindkettőhöz

különböző horgok tartoznak. Ezek a horog függvények az `.onLoad()` és `.onAttach()`. Ugyanazok az argumentumaik, mint a `.First.lib()`-nek, és a névtérben kell definiálni és nem szabad exportálni őket.

A névtérrel rendelkező csomagokhoz viszont használjuk a `.Last.lib()` függvényt. Van egy horog, az `.onUnload()`, ami a névterek kitöltésekor hívódik meg, argumentumként a csomag telepítési helyének teljes elérési útjával. Az `.onLoad()`-ot a névtérben kell definiálni, én nem szabad exportálni, míg a `.Last.lib()`-et exportálni kell.

A csomagok nagyon ritkán használják az `.onAttach()` függvényt. A kódokat opciók beállítására és közös objektumok betöltésére az `.onLoad()` függvénybe tegyük, vagy használjunk `useDynLib()` direktívákat. Lehet egy vagy több ilyen direktíva, ami engedi a betölteni kívánt közös objektum specifikálását a NAMESPACE fájlban. Az `obj_nev` közös objektum regisztrálása a `library.dynam()`-mal való betöltéshez a `useDynLib(obj_nev)` direktívával történik. A regisztrált objektumok betöltése a csomag kódjainak betöltése után, de a horgok betöltés előtt történik. A `useDynLib()` direktívának meg lehet adni azokat a rutinokat is, amiket majd használni szeretnénk az R-ben a közös objektumunkból, a `.C()`, `.Call()`, `.FORTRAN()` és `.External()` interfész-függvényeken keresztül. Ezt a direktívában a rutinok megadásával a csomagnevünk után tehetjük meg. Azzal, hogy ezeket a rutinokat specifikáljuk a `useDynLib()` segítségével, elérjük, hogy a csomagbetöltés során a közös objektumban lévő rutinok neveit feloldja az R, majd a feloldott nevekhez hozzárendel ugyanolyan nevű R változókat, és ezek a változók hozzáadódnak a névtérhez. Például a `useDynLib(peldacsomag,rutin)` névtérfájlba való bevitele és a csomag betöltése után a `.Call("rutin", param1, paramx, PACKAGE="peldacsomag")` függvényhívás leegyszerűsödik a `.Call(rutin, param1, paramx)` függvényhívásra.

Két nagy előnye van az előző megközelítésnek. Az egyik az, hogy szimbólum keresése csak egyszer történik, nem pedig minden alkalommal, amikor meghívjuk a rutint. Másodszor megszüntet minden félreérthetőséget az olyan szimbólumok hozzárendelésekor, amelyek több lefordított könyvtárban is benne lehetnek. Így lehetséges például ugyanazon csomag több verzióját használni egyszerre is ugyanazon R munkaszakaszban.

Előfordulhat, hogy egy R-beli változónak és az előző módon a névtérbe töltött rutinnak ugyanaz a neve. Ennek elkerülésére a `useDynLib()` direktívának megadhatjuk,

hogy a rutinneveknek más változónevet rendelünk, mint a saját neve. Ezt a következő argumentummal lehet megtenni: `alternativ_nev = rutin_nev`.

Előfordulhat olyan eset, amikor nincs értelme az R változók hozzárendelésének a rutinokhoz a névtérben. Megeshet ugyanis, hogy túl költséges ezeknek a kiszámítása a rutinokhoz a betöltődés alatt, ha ezen rutinok nagyon ritkán vannak használva. Ebben az esetben is fenntarthatjuk a hatékonyságot a DLL megfelelő használatával, mindig, amikor a rutint meghívjuk az R-ből. Először hivatkoznunk kell tudni a DLL-re az R-ben. Ehhez definiáljunk egy dinamikus könyvtár változót a névtérben az `enDLLem = useDynLib(csomagnev)` direktíva segítségével. Ezután a kódban már használhatunk dinamikus címzést a $ után:`.Call(enDLLem$rutin, param1, paramx)`. A $ operátor feloldja a megadott nevű rutint a DLL fájlból a `getNativeSymbol()` függvény segítségével. Ez ugyanaz a számítás lesz, mint a sima rutinnév feloldásnál, annyi különbséggel, hogy ez nem a csomag betöltésekor végződik el, hanem a `enDLLem$rutin` esetén.

```
export(hopfield, energy)

importFrom(stats, predict)

S3method(energy, hopfield)
S3method(predict, hopfield)
```

`NAMESPACE` fájl tartalma a csomagomban

5.8. A dokumentálás

Az R objektumok az „R dokumentációs” (Rd) formátumban írott fájlokban vannak dokumentálva. Ez egy egyszerű és hatékony nyelv, amely nagyon hasonló a (La)TeX-hez, és átalakítható sok más formátumba, például HTML-be és LaTeX-be is. A fordítást egy Perl script hajtja végre, ez az `Rdconv` és az `R_HOME\bin` könyvtárban található. Érdekességképpen megjegyezzük, hogy az alap R disztribúcióban több mint 1200 ilyen dokumentációs fájlt lehet találni.

Egy Rd fájl három részre osztható. A fejrész alapvető információkat tárol a fájl nevéről, a dokumentált témákról, egy rövid szöveges ismertetőt és R használati információkat az objektumról. További információkkal a testrész szolgál például a függvény

argumentumairól és a visszatérési értékéről. Végül van egy lábjegyzet rész a kulcsszavakkal. A fejléc és a lábléc kötelező.

Az alapvető utasítások az R objektumok, elsősorban függvények dokumentálására a következők:

`\name{nev}`: Tipikusan az Rd fájl neve, valamint annak az objektumnak a neve, amit dokumentálunk. A névnek egyedinek kell lennie a csomagban.

`\alias{topic}`: Ez az összes „témát” specifikálja, amit a fájl dokumentál. Ezek az információk egy index adatbázisba lesznek gyűjtve, hogy az online súgórendszer ebben kereshessen. A `topic` tartalmazhat szóközöket, de az elejéről és a végéről ezek el lesznek hagyva. Lehet több `alias` bejegyzés is egy fájlban, sokszor ugyanis kényelmes több objektumot egy helyen dokumentálni. Legfőképp azokat, amelyek viselkedése és paraméterezése szinte megegyezik. És az is kényelmes, hogy egy R objektumra többféleképpen is hivatkozhatunk, ugyanis az `alias`-ban nem kell egy létező objektum nevét megadnunk. A név nem feltétlen egy dokumentált téma, viszont ha azt szeretnénk, hogy az legyen, akkor kell egy explicit `alias` bejegyzés is hozzá.

`\title{cim}`: Az Rd fájl címinformációját tartalmazza. Ne legyenek benne speciális karakterek vagy írásjelek, és kezdődjön nagybetűvel.

`\description{szoveg}`: Egy rövid leírás, hogy mit csinál(nak) a dokumentált függvény(ek).

`\usage{fuggv(arg1, arg2, …)}`: Egy vagy több sor a fájlban dokumentált függvény(ek) és az argumentumaik formájáról. Ezek írógép betűtípussal fognak megjelenni a dokumentációnkban. A specifikált `usage` információnak tökéletesen meg kell egyeznie a dokumentált függvény(ek) definíciójával. A `usage`-on belül használhatunk további utasításokat. Generikus metódus leírásához használjuk a `\method{generikus}{osztaly}` utasítást. Ez egy `generikus` generikus függvényt fog jelenteni, amelynek `osztaly` az osztálya. Például a

```
\usage{
\method{predict}{hopfield}(hopfield, adatok)
}
```

így fog megjelenni a dokumentációban:

```
Usage:

     ## S3 method for class 'hopfield':
     predict(hopfield, adatok)
```

`\arguments{…}`: Leírás a függvény argumentumairól. Benne minden argumentumot a `\item{arg_i}{arg_i leirasa}` utasítással írjunk le külön-külön. A bejegyzések előtt és mögött lehetnek leíró szövegek.

`\details{…}`: Itt a `description`-nél megadott rövid leírást bővíthetjük ki sokkal részletesebben.

`\value{…}`: Leírás a függvény visszatérési értékéről. Amennyiben ez egy lista több értékkel, akkor itt is használhatjuk az `item` utasítást a lista minden elemének kifejtésére. A bejegyzések előtt lehet szöveg.

`\references{…}`: Irodalmi referenciák megadását itt tehetjük meg. Az `\url{…}` utasítást használjuk az internetes linkek megadásához.

`\note{…}`: Speciális megjegyzések, amikre fel szeretnénk hívni a figyelmet.

`\author{…}`: Információ az Rd fájl szerzőjéről, használjuk az `\email{…}` utasítás e-mail cím megadására, vagy az `\url{…}`-t esetleges linkek megadására.

`\seealso{…}`: Ide a kapcsolódó R objektumokra hivatkozó mutató kerül. Használjuk a `\code{\link{…}}`-et a hivatkozáshoz. Egy R objektum nevét a `\code{…}` segítségével adhatjuk meg, a `\link{…}` pedig hiperhivatkozást állít elő a kimenetre a megfelelő formában.

`\examples{…}`: Példakódok kerülnek ide, arról, hogy kell használni a függvényt. Szintén írógépformázással fog megjelenni. Az itt megadott kódok nem csak dokumentációs célból hasznosak, hanem tesztkódokként funkcionálnak, az R diagnosztizáló ellenőrzéséhez. Alapból az itt beírt kódok kiíródnak a dokumentációba, és az `R CMD check` is lefuttatja. Ahhoz, hogy ezt elkerüljük, használjuk a `\donotrun{…}` utasítást, ez megjelenik, de nem fut le. A `\donotshow{…}`-ba írt utasítások viszont lefutnak (az `example()` függvény is lefuttatja ezeket), de nem jelennek meg a dokumentációban. A `\donotrun{…}`-on kívül az összes kódnak futtathatónak kell lennie, és itt ne használjunk rendszerfüggő funkciókat. A `\donotrun{…}`-ba írt kódok megjegyzésekként lesznek megjelenítve a dokumentációs

fájlba. A példához szükséges adatok használata elkerülhető véletlen számok generálásával, vagy a standart adatkönyvtárban lévő adatok használatával.

`\keyword{kulcs}`: Minden kulcs az alap kulcsszavakat tartalmazó `KEYWORDS` (`R_HOME\doc`) fájlból felsoroltak közül vesz fel értéket. Legalább egy kulcsszó bejegyzésnek kell lennie, de lehet több is, ha úgy érezzük, hogy az objektumunk több csoportba is beletartozik. Az `internal` speciális kulcsszót akkor írjuk, ha az objektumunk belső használatra való, ekkor ugyanis nem fog megjelenni az API-ban. Ha ilyen objektumra keresünk rá, akkor megjelenik ugyan a súgó róla, de ez az objektum a súgó indexelt listáiban nem fog megjelenni.

A `prompt()` R függvénnyel gyorsíthatjuk meg a dokumentálási munkánkat. A függvényünket paraméterül adva ez a függvény elkészíti az Rd fájlunk vázát, és néhány mezőt ki is tölt helyettünk (például a `usage`-ot és az `arguments`-et). Nekünk szinte már csak a szöveges információk beírása fog maradni.

Az adathalmazokat is dokumentálnunk kell. Az ilyen dokumentációs fájl csak kevéssel tér el a függvények dokumentálásától. Az olyan részek, mint a `\value{…}` vagy az `\arguments{…}`, nem kellenek, helyette az adatok formátuma és a forrása szükséges. Azaz a következő mezők kellenek:

`\docType{…}`: A dokumentált objektum típusát jelenti, adathalmazoknál ennek `data`-nak kell lennie.

`\format{…}`: Leírás az adathalmaz formátumáról, például mátrix vagy vektor-e. Mátrixok és adatkeretek esetén az oszlopokat fejtsük ki bővebben.

`\source{…}`: Részletek az eredeti forrásról, azaz arról honnan származnak ezek az adatok. A másodlagos kiegészítő forrásokat `references`-be írjunk.

Lehetséges a szöveget formázni az előző utasításokon belül, valamint lehetőség van speciális jelentéssel bíró szövegek bevitelére (például e-mail cím). Nem fogok mindent felsorolni, csak pár példát adok: `\bold{text}` a szöveget félkövéren írja ki, `\emph{text}` kurziváltan írja ki a szöveget. A speciálisak közé tartozik például az előzőekben említett `\email{}` vagy a `\url{}` is. A teljes utasításkészlethez lásd az R súgó fájlt.

Táblázatot létrehozni a `\tabular{…}{…}` utasítással lehet. Első paraméterben meg kell adni minden egyes oszlophoz a szöveg igazítását (az `l` balra igazítást, `r` jobbra igazítást,

`c` középre igazítást jelentő betűkkel). A második paraméterként a táblázatot vihetjük be, a sorait `\cr` utasítással, oszlopait `\tab` utasítással elválasztva. Arra figyeljünk, hogy minden sorban azonos számú oszlopot vigyünk be, és mindenhova írjunk valamit (a szóköz megengedett).

Kereszthivatkozást a `\link{topic}` segítségével készíthetünk. A `topic`-nak egy másik súgó fájl `alias`-ához kell tartoznia. A kereszthivatkozást tartalmazó súgó fájlnak nem kell egy csomagban lennie a hivatkozott fájlal. Azt vegyük számításba a kereszthivatkozás megadásánál, hogy míg az `alias`-nál a kezdő és végződő szóköz karakterek el lesznek távolítva, a `link`-nél ez nem lesz megtéve, így könnyen előfordulhatnak hibák.

```
\name{predict.hopfield}
\alias{predict.hopfield}
\title{A bemeno adatokra meghatarozza a halozat kimenetet}
\description{
        A bemeno adatokra meghatarozza a megadott halozat kimenetet. A halozatnak
egy mar definialt Hopfield halonak kell lennie.
}
\usage{
        \method{predict}{hopfield}{object, adatok}
}
\arguments{
        \item{object}{ Egy \code{hopfield} osztalyu objektum, amely egy betanitott
Hopfield halozatot tartalmaz }
        \item{adatok}{ A bemeno adatok vektora vagy matrixa amire ki akarjuk
szamoltatni a halozat kimenetet }
}
\value{
        Ha a bemenet vektor volt akkor a visszateresi ertek a halozat kimenetet
tartalmazo vektor, ha a bemenet matrix volt akkor a kimenet is matrix lesz soronkent a hálózat
kimeneti ertekeivel.
}
\references{ R. Rojas. Neural Networks. Springer-Verlag, 1996. }
\author{ Kovacs Tibor Krisztian }
\seealso{ \code{\link{hopfield}}, \code{\link{energy}} }
\examples{
        hop <- hopfield( matrix(c(-1,1,-1,1,1,-1), nr=2, nc=3 ) )
        hop
        josol <- matrix(c(-1,1,-1,1,1,-1,-1,-1,-1,-1,-1,1), nr=4, nc=3 )
        josol
        predict( hop, josol )
}
\keyword{ neural }
```

Egy dokumentációs fájl a csomagomból (a predict.hopfield() függvényhez)

5.9. Csomagcímkék írása

Az Rd formátumú súgó fájlokon kívül, az R támogatja tetszőleges dokumentumok használatát. Alaphelyzetben ezeknek a dokumentációs fájloknak a helye az `inst\doc` csomagkönyvtárban van, majd a csomag installálása közben ezek átmásolódnak a `doc` R könyvtárba. A csomag súgó fájljaiban lévő mutatók segítségével az installált dokumentumok automatikusan is létrejöhetnek. Az `inst\doc` könyvtárban lévő súgó fájlok ugyan bármilyen formában lehetnek, de a készítők a PDF formátumot ajánlják a hordozhatóság miatt, ezt a fájltípust ugyanis minden rendszeren könnyen meg lehet nyitni.

A speciális dolgok (mint például LaTeX képletek) Sweave formátumban vannak dokumentálva, amit csomagcímkéknek nevezünk. A Sweave lehetőséget nyújt a LaTeX dokumentumok és R kódok integrálására. A Sweave-et az R bármely alapkiadásában megtaláljuk az `utils` alkönyvtárban. Az `inst\doc` könyvtárban talált összes címkét ellenőrzi az `R CMD check`, az összes benne található R kódrészlet futtatásával, hogy biztos legyen a dokumentáció és a kód közötti egyezőség. Az `eval=FALSE` opcióval megadott kódok nem lesznek tesztelve. Az összes címke `R CMD check` tesztelésekor használt munkakönyvtára a már installált csomag `doc` alkönyvtára lesz. Bizonyosodjunk meg róla, hogy az összes olyan fájlt, amire a címkénknek szüksége lehet, elérhetővé tegyük az `inst\doc` forráskönyvtárba másolásával vagy a `system.file()` függvényhívással.

Az `R CMD build` automatikusan létrehozza a csomaghoz szolgáltatott címkékből a PDF fájlokat. Épp ezért a címkék PDF formátumainak csomaghoz adása nem szükséges, mert a címkék lefordíthatóak installáláskor is. Egyetlen kivétel erre az, ha a címke írója használt speciális LaTeX utasításokat, amik nagy valószínűséggel csak a saját gépén találhatóak meg.

Alapbeállításként az `R CMD build` az összes Sweave formátumban lévő fájlon futtatja a `Sweave`-et. Ha nincs megadva `Makefile` az `inst\doc`-ban, akkor az alap paraméterezés a `texi2dvi --pdf` lesz. Amennyiben volt `Makefile`, akkor az `R CMD build` a `Sweave` lépés után próbálja azt végrehajtani. A `Makefile`-nak mindkét PDF létrehozási módszerre gondot kell viselnie, és utánuk fel kell takarítania a szemetet, azaz törölnie kell minden olyan fájlt, aminek nem szabad a végső installált könyvtárban maradnia. A `make` végrehajtása független a Sweave formátumban lévő fájlok jelenlététől.

5.10. Csomag feltöltése a CRAN-ra

A CRAN egy olyan webhely, ahonnan elérhető mind az R környezet, mind pedig a hozzá fejlesztett csomagok. A felhasználók csatlakozhatnak a fejlesztőmunkához, és tulajdonképpen szinte bárki feltölthet csomagot a CRAN-ra. Természetesen ezeket a csomagokat előbb megvizsgálják a fejlesztők, majd az ő jóváhagyásuk után kerülhet publikusan elérhetővé a csomagunk. Éppen az ő munkájuk megkönnyítésére, mielőtt csomagot tennénk fel a szerverre, végezzünk el két ellenőrző lépést. Az egyik az `R CMD check` futtatása lehetőleg minden a csomagunkban elérhető komponens ellenőrzésével. Majd az `R CMD build` futtatásával hozzuk létre a `.tar.gz` tömörített fájlt. Ha az előző kettővel kész vagyunk, feltölthetjük a kapott tömörített fájlt az `ftp://cran.R-project.org/incoming/` ftp címen, felhasználónévként `anonymous`-t jelszóként az email címet megadva.

6. Összefoglalás

A dolgozatom célja az volt, hogy a Hopfield neurális háló implementálása mellett bemutassam az R statisztikai és programozási nyelvet. Úgy gondolom, hogy a Hopfield hálózatot megfelelően definiáltam, a lényeges tételeket felvázoltam, majd a hálózat implementációjának leglényegesebb részeit is megfelelően kifejtettem. Habár az R nyelvet nem definiáltam pontosan, csak kóstolót nyújtottam a sokszínűségéből és a benne rejlő lehetőségekről, de a dolgozatomnak ez nem is volt célja hely és időhiány miatt. A csomagokról is csak elnagyoltan sikerült írnom. A főbb részeket úgy gondolom megfelelően sikerült kifejtenem, de az apróságokra nem jutott idő, így lehetnek ebből kifolyólag homályos részek. Éppen a néhol elnagyolt részletek miatt, a dolgozatomon sokat lehetne bővíteni, sok kimaradt részt bele lehetne írni. Mindezen tények ellenére meg vagyok elégedve a dolgozatom hosszúságával, sikerült jól tömöríteni a hatalmas témakört.

Végül köszönetet szeretnék mondani Jeszenszky Péternek, a szakdolgozatom elkészítése alatt felém nyújtott türelméért és segítőkészségéért.

7. Irodalomjegyzék

[1] The R Project for Statistical Computing. `http://www.r-project.org/`.

[2] The R Development Core Team. *An Introduction to R*. `http://cran.r-project.org/doc/manuals/R-intro.html`.

[3] The R Development Core Team. *R Language Definition*. `http://cran.r-project.org/doc/manuals/R-lang.html`.

[4] The R Development Core Team. *R Installation and Administration*. `http://cran.r-project.org/doc/manuals/R-admin.html`.

[5] The R Development Core Team. *Writing R Extensions*. `http://cran.r-project.org/doc/manuals/R-exts.html`.

[6] Michael J. Crawley. *The R book*. Wiley, 2007.

[7] R. Rojas. *Neural Networks*. Springer-Verlag, 1996.

[8] Haykin, Simon S. *Neural networks: a Comprehensive Foundation. 2nd ed*. Prentice Hall, 1999.

[9] Horváth Gábor (szerkesztő). *Neurális hálozatok és műszaki alkalmazása*. Műegyetemi Kiadó, 1998.