Természetes nyelvek gépi tanulása – az ABL algoritmus

Diplomamunka

Írta: Bartók Gábor

Matematikus szak

Témavezető:

Dr. Szepesvári Csaba
Gépi tanulás csoport
MTA, SZTAKI

Belső konzulens:
Grolmusz Vince
Egyetemi docens
Számítógéptudományi tanszék
Eötvös Loránd Tudományegyetem, Természettudományi Kar

Tartalomjegyzék

1. Bevezetés **4**

2. Nyelvfeldolgozó algoritmusok **6**

2.1. Statisztikai elemzés – szópárok keresése 7

2.2. Szavak jelentés szerinti csoportosítása 10

2.3. Leghosszabb gyakori minták keresése – az ADIOS algoritmus . 12

2.4. Szavak morfémákra tagolása 13

3. Az ABL algoritmus **16**

3.1. Célok 16

3.2. Áttekintés 17

3.3. Alignment Learning 17

3.3.1. Az Edit Distance algoritmus 18

3.3.2. Típusok összefűzése 20

3.4. Selection Learning 21

3.4.1. Részmondatok gyakorisága 22

3.4.2. Részmondat-halmazok „jósága” 22

3.4.3. Az algoritmus 23

3.5. Összefoglalás 24

3.6. Eredmények 25

3.6.1. Korpuszok 25

3.6.2. Mértékek 25

4. Problémafelvetések **28**

4.1. Típusösszevonások 28

4.2. Futási idő 30

4.2.1. Megvalósítás 31
4.3. Általánosított részmondatok 32
4.4. Részmondatok súlyozása 32
4.5. Iteráció 33

5. Tesztek, eredmények **34**
5.1. Az algoritmus gyorsítása 35
5.2. A lépések megfordítása 39

6. Összefoglalás **42**

1. fejezet

Bevezetés

Korunkban a számítógép használata életünk mindennapos részévé vált. Ha valamit el akarunk intézni, ha elutazunk, ha jegyet rendelünk, de ha csak bevásárolunk, ha zenét hallgatunk, biztosan kapcsolatba kerülünk számítógéppel. Természetes módon merül fel az igény, hogy a számítógép nyelvét közelebb hozzuk az ember nyelvéhez.

A számítógépnek nyelvi téren is széleskörű alkalmazási lehetőségei vannak: beszédfelismerés, helyesírásellenőrzés, szöveges utasítások végrehajtása, szövegek feldolgozása és rendszerezése, stb. Ehhez olyan adatbázisokat kell építenünk, melyek nyelvtani szabályokkal, vagy gyakori mintákkal segítenek a nyelvfeldolgozásban. Ilyen adatbázisok kézzel való létrehozása rendkívül időigényes, ezáltal meglehetősen drága folyamat. Célunk, hogy ezeket az adatbázisokat is számítógéppel tudjuk létrehozni. Meg kell tehát tanítanunk a számítógépet a nyelv összefüggéseire, szabályaira. Olyan tanulóalgoritmusra van szükség, mely inputként mondatokat kap, minden további információ nélkül. Ezekből kiszűri az összefüggéseket, mintákat. A cél, hogy az elemzés után az algoritmus képes legyen eddig nem látott mondatokat is ellenőrizni, illetve szintaktikailag helyes mondatokat felépíteni. Az algoritmusoknak ezen fajtáját hívjuk *önszervező tanulás*nak (unsupervised learning).

Az önszervező tanulóalgoritmusok egyike a 2001-ben Menno M. van Zaanen által publikált ABL algoritmus (Alignment-Based Learning – „Illeszkedés alapú tanulás”[1]. Dolgozatomban ennek az algoritmusnak a működését, valamint az ebben felmerülő problémákat, fejlesztési lehetőségeket fejtem ki. A következő fejezetben néhány másik tanulóalgoritmust fogok röviden le-

írni, az azután következő fejezetekben pedig kizárólag az ABL algoritmussal foglalkozunk.

2. fejezet

Nyelvfeldolgozó algoritmusok

Amióta a számítógépes nyelvfeldolgozás tudománya fejlődésnek indult, nagyon sok és nagyon sokféle algoritmus született. Ezekben az algoritmusokban leginkább az a közös, hogy mindegyik nyelvtani szabályokat vagy más, mondatalkotást vagy elemzést segítő adatbázisokat akar előállítani. Ezen belül azonban a célok különbözhetnek. A kimenet jellege természetesen függ a felhasználói igénytől. Néhány példa:

- Szó n-esek (ún. n-gramok), melyek gyakran fordulnak elő a nyelvben (ezek lehetnek szószerkezetek, vagy gyakori kifejezések)
- Környezetfüggetlen nyelvtani szabályok (CFG – *Context Free Grammar*)
- Sztochasztikus környezetfüggetlen nyelvtani szabályok (SCFG): minden nemterminálishoz hozzárendelünk egy valószínűségi eloszlást, ami azt mutatja, hogy az adott nemterminálishoz milyen valószínűséggel tartozik egy-egy szabály jobb oldala.

A különbözőség ellenére fontos, hogy legyen olyan tesztelési módszer, amivel többé-kevésbé össze lehet hasonlítani a különböző eljárásokat, illetve jellemezni lehet a programok hatékonyságát. Az egyik ilyen ellenőrző módszer az ún. *bejóslás*, vagy predikálás. Ennek is több változata létezik, de mindegyiknek az a lényege, hogy miután az algoritmus túl van a tanulási fázison, kap egy olyan mondatot vagy kifejezést, aminek hiányzik a vége. Ezen kívül még elképzelhető, hogy kap olyan szavaknak egy listáját, melyek közül választania

kell: melyik illik vagy melyek illenek a mondat végére. Az is lehet, hogy nem kap listát, hanem az egész szókincsből kell választania. A teszt egy gyengébb változatában elég, ha a hiányos mondat feldolgozása után kimenetként kiad két szóhalmazt: az egyik a lehetséges szavak listája, a másik azoké a szavaké, melyek biztosan nem illeszthetők a mondat végére. Ezek után megvizsgálhatjuk, hogy az adott algoritmus az esetek hány százalékában dönt helyesen.

Ebben a fejezetben néhány ilyen algoritmust írok le vázlatosan. Igyekeztem, hogy legyenek köztük egymásra építőek és egymástól lényegesen különbözők is.

2.1. Statisztikai elemzés – szópárok keresése

Az Ido Dagan, Lilian Lee és Fernando C. N. Pereira által írt cikk (Similarity-Based Models of Word Cooccurrence Probabilities)[2] egyes szavak egymás utániságának gyakoriságával foglalkozik. A tréning korpuszból kiszámolja a szópárok valószínűségét, aztán ezen információk segítségével végzi a bejóslás műveletet.

Ezt a módszert könnyen általánosíthatjuk: szópárok helyett viszgálhatjuk n-gramok előfordulási gyakoriságát. Így pontosabb képet kaphatunk a mondatok szerkezetéről, ugyanakkor a képletek nagyon elbonyolódnak, miközben az algoritmus alapelve ugyanaz marad. Ezért itt csak szópárokkal fogunk foglalkozni.

A probléma az, hogy a korpuszban nem szereplő szópárok 0 valószínűséget fognak adni, tehát úgy fogja a program elkönyvelni, hogy ilyen szópár sohasem fordul elő. Márpedig nem lehet akkora méretű korpuszt adni inputnak, hogy minden a valóságban lehetséges szópár előforduljon benne.

Az sem segít, ha a korpusz nagyon nagy: egy megfigyelés („tétel”) szerint egy n szóból álló korpuszban nagyságrendileg $n^{\frac{2}{3}}$ különböző szó van. Tehát ha növeljük a korpusz méretét, új szavak is meg fognak jelenni. A szavak számáról bővebben:[3]

Ennek kiküszöbölésére több megoldást ad a cikk, példával illusztrálva, hogy melyik megoldásnak mik az előnyei és hátrányai. Mindegyik módszer

fő eszköze, hogy elsimítja (smoothing) a valószínűségeket, a nulla gyakoriságú szópárok valószínűségét megnöveli, miközben a magas gyakoriságoktól „elvesz”.

A legegyszerűbb megközelítés, hogy a sosem látott szópárok valószínűségét a szavak korpuszbeli előfordulási gyakoriságával számítjuk ki. Ez a *back-off* módszer (Slava M. Katz, 1987) [4]. A legfőbb gond a módszerrel az, hogy a nyelv sajátosságai miatt lehetséges, hogy két rendkívül gyakori szó nagyon ritkán fordul elő egymás mellett. Az ilyen példáknál a módszer hibázni fog (magyarul egy ilyen példa: *a az*).

Másik megoldás, hogy a sosem látott szópár első eleme helyett ahhoz *hasonló* szavakat párosítunk, és ezekből számoljuk ki az eredeti szópár valószínűségét. Kell tehát egy hasonlósági függvény szavak között. Ha ez megvan, akkor a szópár valószínűségét a következő képlet adja:

$$P_{SIM}(w_2|w_1) = \sum_{w_1^{'} \in S(w_1)} \frac{W(w_1, w_1^{'})}{norm(w_1)} P(w_2|w_1^{'})$$

A jelölések: $P(w_2|w_1)$ a (w_1, w_2) szópár valószínűsége, feltéve, hogy az első szó w_1, $W(w_1, w_1^{'})$ a hasonlósági függvény, S_w a w-nek egy a hasonlósági függvény szerinti környezete, $norm(w_1) = \sum_{w_1^{'} \in S(w_1)} W(w_1, w_1^{'})$ csak normalizálja az összeget.

Az S_w környezet sugarát mi határozhatjuk meg. Minél nagyobbnak választjuk, annál messzebb lévő szavak is befolyásolni fogják a végeredményt. Ha kicsinek választjuk, akkor csak a nagyon hasonló szavak fognak szerepelni a képletben.

Hátra van még a $W(w_1, w_1^{'})$ hasonlósági függvény konstruálása. Ehhez először egy definícióra van szükségünk.

1. Definíció. *Kullback-Leibler(KL) divergencia.*

$$D(w_1||w_1^{'}) = \sum_{w_2} P(w_2|w_1) \log \frac{P(w_2|w_1)}{P(w_2|w_1^{'})}$$

$D(w_1||w_1^{'})$ egy nemnegatív kétváltozós függvény, ami akkor és csak akkor 0, ha $w_1 = w_1^{'}$, viszont sajnos nem szimmetrikus és nem teljesül a háromszög-egyenlőtlenség sem, úgyhogy nem nevezhetjük metrikának. Sőt, értelmezni is csak akkor tudjuk, ha a fenti valószínűségek egyike sem nulla. Ennek eléréséhez megint alkalmaznunk kell a valószínűség finomítási technikát (a 0

értékeket kicsit megnöveljük a magas értékek rovására). Ezzel együtt viszont a függvény kiszámítása nagyon sok időt igényel, főleg ha a korpusz szókincse nagy.

A KL-divergecia szemléletes jelentése a következő: Képzeljük el, hogy egy hírforrásból jelsorozatok érkeznek! Tegyük fel, hogy a jelsorozaton belül valamilyen eloszlás szerint követik egymást a jelek. Ha ismerjük az eloszlást, akkor a jelsorozatokat képesek vagyunk kevesebb jellel tömöríteni. Ha viszont nem ismerjük, az eddigi jelsorozatok alapján készíthetünk egy becsült eloszlást. A KL-divergencia azt adja meg, hogy a becsült és az eredeti eloszlás különbözősége okán az optimálisnál hány bittel kell többet használnunk a jelsorozatok kódolásához

Ez a függvény a fenti szépséghibák ellenére kiválóan alkalmas hasonlósági függvények előállítására. Ha már egyszer kiszámoltuk ezeket az értékeket, az algoritmus folyamán többször használhatjuk, nem kell mindig újraszámolni. Ezek után megadhatjuk hasonlósági függvények egy családját:

$$W_D(w_1, w_1^{'}) = 10^{-\beta D(w_1||w_1^{'})}$$

A β érték hangolásával módosíthatjuk a függvény viselkedését: ha β magas, akkor csak a w_1-hez nagyon közeli szavak fogják jelentősen befolyásolni az eredményt, ha alacsony, akkor távolabbi szavak is számítani fognak.

A KL divergencia hibáinak egy részét küszöböli ki a

2. Definíció. *Jensen-Shannon divergencia:*

$$J(w_1, w_1^{'}) = \frac{1}{2}\left[D\left(w_1||\frac{w_1 + w_1^{'}}{2}\right) + D\left(w_1^{'}||\frac{w_1 + w_1^{'}}{2}\right)\right]$$

A KL divergencia nemnegatívságából következik, hogy ez a függvény szintén nemnegatív. Sőt, a Jensen-Shannon divergencia ráadásul szimmetrikus is. És van egy ennél sokkal fontosabb tulajdonsága is, a könnyű kiszámíthatóság:

$$J(w_1, w_1^{'}) = H\left(\frac{p + p^{'}}{2}\right) - \frac{1}{2}H(p) - \frac{1}{2}H(p^{'})$$

ahol $p^{(')}(w_2) = P(w_2|w_1^{(')})$, $H(q) = -\sum_w q(w)\log q(w)$ tetszőleges q diszkrét eloszlás *entrópiája*.

Ugyanúgy, ahogy a KL divergenciánál, itt is definiálhatjuk a hasonlóságfüggvényeket: $W_J(w_1, w_1^{'}) = 10^{-\beta J(w_1, w_1^{'})}$.

Egy másik lehetőség a hasonlóságfüggvény előállítására az L_1 norma:

3. Definíció.

$$L(w_1, w_1^{'}) = \sum_{w_2} |P(w_2|w_1) - P(w_2|w_1^{'})|$$

Ezzel rögtön megadható a hasonlóságfüggvény: $W_{L(w_1, w_1^{'})} = (2 - L(w_1, w_1^{'}))^{\beta}$

2.2. Szavak jelentés szerinti csoportosítása

Distributional Clustering of English Words (Fernando Pereira, Naftali Tishby, Lillian Lee)[5]. Ez az algoritmus matematikai apparátusát tekintve erősen hasonlít a fentebb leírthoz. Itt is szópárok gyakoriságát vizsgáljuk, konkrétabban alany-állítmány kapcsolatokat. Az igazi különbség az előző algoritmushoz képest az, hogy egészen más a célja. Miért van szükség a szavak jelentés szerinti megkülönböztetésére? A felhasználási lehetőségek messzire vezetnek.

Például ha olyan internetes keresőt szeretnénk üzemeltetni, ahol nem kulcsszavak szerint keresünk oldalakat, hanem a felhasználó *kérdéseket tesz fel*, és erre kéne válaszolnia a számítógépnek. Egy egyszerű példamondat:

Milyen színű az alma?

Ha erre a kérdésre akarunk válaszolni, fel kell ismernünk, hogy a válasz, amit a felhasználó vár, egy *szín*. Ahhoz hogy tudjunk válaszolni, szükségünk van egy szóhalmazra, amiről tudjuk, hogy elemei színek. Ha ez megvan, már könnyen válaszolhatunk: megkeressük, hogy ebből a halmazból melyik az a szó, ami a leggyakrabban szerepel együtt az *alma* szóval. Ha megtaláltuk, kiírhatjuk a választ:

Piros.

Nézzük a következő kérdést:

Az alma vörös?

Az előző kérdésből megtudtuk, hogy az alma piros, most arra kéne válaszolni, hogy a színek halmazában a piros és a vörös elég közel van-e egymáshoz ahhoz, hogy mindkettőt használni lehessen ugyanabban a helyzetben.

Tehát a szavakat nem elég egyszerűen csoportosítani, hanem több szinten kell tárolni. minél lejjebbi szinten vagyunk, annál kisebbek lesznek a halmazaink.

Következő példánkban nem kérdést akarunk feltenni, hanem kulcsszó szerint akarunk keresni. Mondjuk merevlemezt szeretnénk vásárolni, és beírjuk a keresőbe, hogy „winchester". Akkor lenne igazán jó a keresőprogramunk, ha nemcsak azokat az oldalakat találná meg, melyekben szerepel a winchester szó, hanem azokat is, melyekben megtalálható a „hard drive" vagy a „HD".

Ilyen típusú problémák megoldására nyújt segítséget az alábbi algoritmus.

Első lépésben minden (v, n) szópárra – ahol v ige (verb), n főnév (noun) – definiálunk egy f_{vn} értéket, ami a megfelelő szópár gyakorisága a korpuszban. Ezekből az értékekből meg lehet határozni a feltételes valószínűségeket:

$$p(v|n) = p_n(v) = f_{vn} / \sum_v f_{vn}$$

Ezek után ismét a KL divergencia segítségével lehet a szavak hasonlóságát vizsgálni. A különbség az előző algoritmushoz képest, hogy nem a szópárok valószínűségének kiszámítása a cél, hanem a főnevek ún. *szoft-klaszterezése*. Ez az eljárás annyiban különbözik más klaszterezési módszerektől, hogy egy elemet nemcsak egy klaszterbe lehet betenni. Ehelyett meg lehet adni, hogy az egyes elemek hányadrészben vannak az egyes klaszterekben.

Az algoritmus által használt eljárás neve *Hierarchical Distributional Clustering*. Lényege, hogy minden iterációs lépésben tovább osztja a már meglévő klasztereket. Így minél többször iterálunk, annál több és annál kisebb klaszterein lesznek. Először csak a szintaktikailag másképp használt szavak fognak elkülönülni egymástól, később azonban megjelennek olyan kisebb klaszterek, melyek látványosan a szavak jelentése szerint különültek el.

Mivel ez az algoritmus nem mondatokat használ fel a tanuláshoz, hanem főnév-ige kapcsolatokat, nem lehet úgy futtatni, hogy nem elemeztük magunk a példamondatokat. Ez eltér az önszervező tanulás alapelvétől. Ennek ellenére érdemes itt megemlíteni az eljárást. Ha ugyanis az algoritmus elé teszünk egy olyan eljárást, ami aránylag kevés hibával ki tudja szűrni a mondatokból az alany-állítmány szerkezeteket, akkor máris eltűnik az emberi beavatkozás a tanulási folyamatból. ilyen elemző eljárás készítése pedig nem reménytelen feladat.

2.3. Leghosszabb gyakori minták keresése – az ADIOS algoritmus

A Zach Solan, David Horn, Eytan Ruppin és Shimon Edelman által 2005-ben publikált algoritmus[6] teljesen új módszereket használ, melyek alapjaiban különböznek az eddig leírtaktól. Előnye, hogy nem csupán szópárokat, vagy általánosabban szó n-eseket tud vizsgálni, hanem automatikusan megtalálja a mondatban a leghosszabb mintákat. Az algoritmus egy szemléletes leírása a következő:

Képzeljünk el egy gráfot, melynek csúcsai a korpuszban előforduló szavak és még két kitüntetett pont – *begin* és *end*. A korpusz minden mondatához hozzárendelhetünk egy *begin-end* irányított utat a gráfban. Az általunk létrehozott objektum annyiban különbözik a megszokott gráfoktól, hogy amikor behúzunk egy utat, az éleket felcímkézzük a mondat sorszámával. Így ha például egy szópár sok mondatban fordul elő, akkor a két szónak megfelelő csúcsok között sok irányított él fog menni, de mind különböző címkével. Egy élsorozatot csak akkor fogunk útnak tekinteni, ha az élek címkéi megegyeznek.

Az algoritmus első lépése az ún. MEX (Motif Extraction procedure): Minden mondatra megvizsgáljuk, hogy a hozzá tartozó irányított út pontjaiban mennyi a kilépő, a belépő és az áthaladó utak száma. Ezek után mindkét irányban végighaladunk az úton, és kiszámoljuk a kilépő (visszafelé a belépő) utak és az áthaladó utak arányát. Ahol az előremenetben számolt arány szignifikáns mértékben lecsökken (itt fellép egy hangolásra alkalmazható paraméter), ott lesz vége a mintának (significant pattern), ahol a visszafelé számolt arány csökken le, ott kezdődik a minta. Ha megtaláltuk a szignifikáns mintát, felveszünk a gráfba egy új csúcsot, ami már nem egy szót fog jelenteni, hanem ezt a mintát. Új éleket is húzunk be: Minden olyan úton, ami végigment a mintán, az átmenő éleket töröljük, a bemenő és kimenő élek helyett pedig behúzunk egy bemenő és egy kimenő ugyanolyan címkéjű élt az új csúcson át. Ezzel elkönyveltük a mintát mint egy „szófordulatot”.

A másik fő lépés az *Általánosított keresőút* (Generalized Search Path). Ebben a lépésben ún. *slot*okat keresünk: olyan utakat, melyek egy adott helyen elágaznak, és rögtön vissza is térnek. (Pl. ha van húsz majdnem ugyanolyan mondat, melyek csak abban térnek el egymástól, hogy más jelzővel van illetve

az alany, akkor a sok melléknév együttesen egy slot.) A keresést úgy végezzük, hogy minden úton végigvezetünk egy előre meghatározott méretű „ablakot", és figyeljük, hogy találunk-e olyan szósorozatokat, melyeknek az eleje és vége megegyezik. Ha megtaláltunk egy slotot, akkor megint új csúcsot vezetünk be, és értelemszerűen behúzzuk az új éleket.

Lényegében ezt a két lépést felváltva végrehajtva addig iterálunk, amíg már nem találunk új szignifikáns mintákat. Ha végeztünk, a kétfajta általunk létrehozott csúcsok segítségével nyelvtani szabályokat készíthetünk: ez lesz az algoritmus kimenete.

2.4. Szavak morfémákra tagolása

Dolgozatom eddigi részében kizárólag olyan algoritmusokról esett szó, melyek mondatok szintaktikai elemzésére voltak alkalmasak. Van azonban a nyelvfeldolgozásnak egy másik nagyon fontos ága: a szóelemzés.

A *morféma* a nyelv legkisebb értelemmel bíró egysége. Egy szó tehát morfémákra bontható. Néhány példa:

meg/rend/el/ed: igekötő – szótő – képző – személyrag

személy/igaz/ol/vány: szótő – szótő – képző – képző

A szóelemző algoritmusok felhasználási területe hasonló a mondatelemző algoritmusokéhoz. Annyit érdemes megemlíteni, hogy a szóelemzés nagyban segítheti a mondatelemzést, kiváltképp az olyan nyelveknél, melyekben gyakori a ragozás, esetleg egy szóhoz több toldalékot is lehet kötni. Ilyenek az ún. agglutináló nyelvek (pl. a magyar is).

A probléma hasonló a mondatelemzéshez: szósorozatok kifejezésekre tagolása helyett betűsorozatokat kell morfémákra tagolni. Kínálkozik az a megoldás, hogy az eddig mondatelemzésre használt algoritmusokat kis változtatással alkalmazzuk szóelemzőként. Míg a mondatelemzés alapegysége a szó volt, a szóelemzés egysége a betű lesz. Emellett természetesen készültek olyan algoritmusok is, melyek kifejezetten morfológiai elemzésre vannak optimalizálva.

Ezek egyike a Delphine Bernhard által publikált algoritmus[7], amit itt röviden vázolok.

Az algoritmus alapfeltevése, hogy egy szó négyfajta morfémából áll elő: *szótő, előtag, utótag* és *kötőelem*. Célunk ezeknek az elemeknek felkutatása az egyes szavakban.

Ebben az algoritmusban találhatunk hasonlóságot az ADIOS és az következő fejezetekben tárgyalt ABL algoritmussal is. Először is specifikáljuk az inputot! Az input most szavak egy L halmaza, ahol minden szó csak egyszer fordul elő. Ebből következik, hogy az egyes szavak nyelvbéli gyakoriságával nem foglalkozunk. Alapfeltevés, hogy minél hosszabb egy szó, annál valószínűbb, hogy szétdarabolható morfémákra. Akkor van tehát jó esélyünk a szóelemzésre, ha előbb a hosszabb szavakkal foglalkozunk.

Az első lépés tehát adott: rendezzük a szavakat hosszúság szerint (mégpedig fordítva – a leghosszabbal kezdve)! Az első (n hosszú) szóra most kiszámítunk néhány értéket:

$$F(k) = \frac{\sum_{i=0}^{k-1} \sum_{j=k+1}^{n} max\left[p\left(s_{i,k}|s_{k,j}\right), \left(s_{k,j}, s_{i,k}\right)\right]}{k * (n-k)}$$

Jelmagyarázat:

- $s_{i,j}$ az i-diktől a j-dik betűig tartó *szubsztring*.
- $k \in [1, \ldots, n]$ egész. (Szemléletes jelentése: a k-dik és a $k+1$-dik betű között.)
- $p\left(s_{i,k}|s_{k,j}\right) = \frac{f(s_{i,j})}{f\left(s_{k,j}\right)}$ és $p\left(s_{k,j}|s_{i,k}\right) = \frac{f(s_{i,j})}{f\left(s_{i,k}\right)}$, ahol $f\left(.\right)$ az adott szubsztring előfordulásainak száma az L korpuszban.

Ezt az F függvényt vizsgáljuk: megkeressük a lokális minimumhelyeit. Ezek lesznek a morfémák határai.

Ha megtörtént a tagolás, most el kell döntenünk, hogy mely morfémák milyen típusúak. Először a szótövet keressük meg: a tagolás után maradt szubsztringek közül a leghosszabb és legkevésbé gyakorit elkönyveljük mint szó tövet, ha legalább kétszer előfordul még a korpuszban, és legalább egyszer valamelyik szó elején. Az ez előtti illetve utáni morfémát előtagként (utótagként) kezeljük, ha rövidebb és gyakoribb, mint a szótő. Az egybetűs előtagokat elhagyjuk, mert az gondokat okozhat a későbbiekben.

Az algoritmusnak ezt a részét addig folytatjuk, amíg a megtalált elő- és utótagok már olyan sokan vannak, hogy a most megtanultaknak már több mint a fele benne van az eddig megtaláltak halmazában.

A következő lépésben elvégezzük a szavak szegmentálását. Összehasonlítjuk az ugyanolyan szótövű szavakat, és a szótő előtti illetve utáni részekben próbálunk olyan szubsztringeket keresni, melyek jelen vannak az eddig felépített elő- és utótagszótárunkban.

Lényegében tehát az algoritmus két lépését így lehet összefoglalni: Először olyan szavakat keresünk, melyekben majdnem biztosan meg tudjuk állapítani, hogy mi a szótő, és hogy mik a szó további összetevői. Ezekből kinyerünk egy morfémahalmazt. A második lépésben annak a morfémahalmaznak a segítségével keressük meg az összes szó elemzését.

3. fejezet

Az ABL algoritmus

Ebben a fejezetben részletesen leírom az ABL algoritmus működését abban a formájában, ahogy azt Zaanen PhD dolgozatában olvastam. Mivel abban a dolgozatban nem minden részfeladat megoldása van pontosan leírva, vannak olyan elemei a fejezetnek, melyeket saját elgondolásom alapján egészítettem ki, így kissé eltérhetnek az eredeti programtól. Ugyanakkor az általam leprogramozott algoritmus működésének fontos pontjai biztosan megegyeznek az eredetivel.

3.1. Célok

Az ABL algoritmusnak fő célja, hogy megtalálja és csoportosítsa a mondatokban lévő kifejezéseket és szintaktikai egységeket.

Az program kimenetét kétféleképpen lehet rendszerezni: Egyrészt kapunk kifejezéseknek egy csoportosított listáját, ahol az egy csoportban lévő kifejezések hasonló funkciót töltenek be a mondatban (pl. időhatározó). De nézhetjük mondatonként is – az algoritmus megmutatja, hogy a mondatban mely szósorozatok tartoznak össze és alkotnak egy kifejezést. Így tulajdonképpen megkapjuk a mondatoknak egy szintaktikai elemzését. Ezt az elemzést úgy is hívják, hogy a mondat *zárójelezése*: a kifejezések határait zárójelekkel jelöljük, pl.: *(Ma délután) (elmegyünk) (egy étterembe).*

Ilyen elemzések után kapunk olyan sablon mondatokat, amikből általánosítva lehetőségünk van új, a korpuszban nem szereplő mondatokat előállítani. Ha például egy meglévő mondatban egy kifejezést kicserélünk egy vele azo-

nos csoportban lévővel, jó esélyünk van, hogy az új mondat értelmes lesz. Ha pedig egy ismeretlen mondatról el akarjuk dönteni, hogy szintaktikailag helyes-e, megpróbálunk benne kifejezéseket keresni, és megnézzük, hogy van-e olyan mondat a korpuszban, amiben ugyanilyen szerkezetet követnek a kifejezések.

3.2. Áttekintés

Az algoritmus azt az alapelvet használja, hogy ha egy szó (vagy egy szósorozat) egy adott mondatban helyettesíthető egy másikkal, akkor a két szó(sorozat) valamilyen szempontból hasonló. Ennek az elvnek az alapját Zellig Harris tette le (Harris's notion of substitutability). Az eredeti definíció így szól:

4. Definíció. *$a = b$ akkor és csak akkor, ha minden c, d láncra cad és cbd egyszerre elemei (illetve nem elemei) a nyelvnek, azaz ha a és b disztribúciója ugyanaz.*

Mi ennek az elvnek egy finomított változatát használjuk: a szavak annál jobban hasonlítanak egymásra, minél több esetben tudjuk kicserélni őket.

5. Definíció. *Az alábbiakban az egy vagy több szomszédos szóból álló sorozatot* részmondat*nak fogom nevezni.*

Az algoritmus két fő részből áll. Az első részben (alignment learning) minden mondatpárt megvizsgálunk, és megkeressük azokat a részmondatokat melyek kicserélhetőek egymással. A második részben (selection learning) minden mondatban keresünk egy – bizonyos szempontból jó – átfedésmentes részmondat-rendszert. Mindkét résznek több változata van. Az alábbiakban ismertetem a különböző megközelítéseket.

3.3. Alignment Learning

Ebben a részben tehát az a feladatunk, hogy a mondatokat páronként összehasonlítsuk. Ahhoz, hogy megtaláljuk azokat a részeket, melyeket kicserélhetünk, elsőként meg kell találnunk a két mondatban *megegyező* szavakat. Ha

ezeket megtaláltuk, könnyen megtalálhatjuk a kicserélhető részmondatokat is: A két mondatot egymás alá írjuk, az egyező szavaknál „összekapcsoljuk". A kapcsolódások közötti részmondatokat kicserélhetőnek tekinthetjük. Akkor van gond, ha a kapcsolódási pontok nem egyértelműek. Pl.:

... from England to Sesame Street

... from Sesame Street where Big Bird lives to England

vagy

... from England to Sesame Street

... from Sesame Street where Big Bird lives to England

Ilyen példák elég nagy korpusz esetén biztosan előfordulnak. Ezekben az esetekben alapvetően két lehetőségünk van. A különböző kapcsolódások közül kiválasztjuk azt, amelyik „a legjobban tetszik", vagy kielemezzük a mondatpárt az összes lehetséges kapcsolódás alapján.

3.3.1. Az Edit Distance algoritmus

Az első esetben tehát meg kell találnunk az optimális kapcsolódást. Ehhez lesz segítségünkre az Edit Distance algoritmus, amely eldönti, hogy hogyan juthatunk el legolcsóbban egyik mondatból a másikba.

Költségfüggvény

Amikor meg akarunk változtatni egy mondatot, háromféle műveletet használhatunk: *beszúrás, törlés* és *csere*. Ha ezeknek a műveleteknek meghatároztuk a költségét, akkor megtudjuk mondani, hogy melyik kapcsolódás jár a legkisebb költséggel.

Zaanen dolgozatában erre a függvényre két változat van leírva: Az „egyszerű" (default) és a „bonyolított" (biased).

Az egyszerű esetben $\gamma(csere) = 2$, $\gamma(törlés) = \gamma(beszúrás) = 1$.

Nézzük meg a fenti példamondatokon, hogy a kétfajta kapcsolásnak mennyi a költsége! Az első esetben töröltük az *England to* részmondatot, majd beszúrtuk a *where Big Bird lives to England*-ot. a Változtatás ára: $2*1+6*1 =$

8. A másodikban $7 * 1 + 3 * 1 = 10$. A fenti két összeillesztésen kívül még jópár lehetőségünk van. Az algoritmus az első esetet találja meg, tehát 8-nál nem lehet kevesebb a költség.

Ezzel a függvénnyel az a probléma, hogy nem bünteti azokat a kapcsolódásokat, ahol a kapcsolt szó a két mondatban nagyon távol van egymástól. Pl.:

... from England to Sesame Street

... from Sesame Street where Big Bird lives to England

Ennek az illesztésnek is 10 az ára, de érezzük, hogy most valahogy mégis közelebb vannak egymáshoz.

A probléma kiküszöböléséhez változtassunk egy kicsit a γ függvényen! Vezessünk be költséget egyezés esetére is, és legyen annál nagyobb, minél messzebbi szavakat akarunk összekapcsolni. Ez lesz a „bonyolított" eset.

$$\gamma(egyezés) = |\frac{x_s}{|s|} - \frac{x_t}{|t|}| * \frac{|s| + |t|}{2}$$

ahol s és t a két mondat, x pedig a szó helye a megfelelő mondatban.
Ha az új függvénnyel futtatjuk a programot, a következő illesztést fogja megtalálni:

... from England to Sesame Street

... from Sesame Street where Big Bird lives to England

Ez az algoritmus már nem kapcsolja össze az egymástól távol lévő szavakat, viszont általában kevesebb egyezést talál. Így szegényebb lesz a kicserélhető részmondatokból álló adatbázisunk.

Az algoritmus

Most már tudjuk, mi alapján keressük a megfelelő illesztést, nézzük hogyan találjuk meg!
A problémát dinamikus programozással fogjuk megoldani. Első lépésként ki kell számolnunk a két mondat távolságát. Ehhez felépítünk egy ($|s| +$

$1) * (|t| + 1)$-es mátrixot. A mátrix i-dik sorának j-dik eleme ($i, j = 0..$) egyenlő annak a két mondatnak a távolságával, melyeket úgy kapunk, hogy a megfelelő mondatoknak csak az első i (j) szavát vesszük. Nyilvánvalóan $M_{0,0} = 0, M_{0,k} = k * \gamma(beszúrás), M_{k,0} = k * \gamma(törlés)$.
A mátrix többi elemét a következőképpen határozzuk meg:

$$M_{i,j} = \min(M_{i-1,j-1} + \gamma(s_i \rightarrow t_j), M_{i-1,j} + \gamma(törlés), M_{i,j-1} + \gamma(beszúrás))$$

Látható, hogy ezzel a számolással tényleg a megfelelő mondatok közti távolságot számoljuk ki. Végül a két mondat távolsága = $M_{|s|,|t|}$.

Ahhoz, hogy előállítsuk a két mondat kapcsolódását, meg kell nézzük, hogy pontosan milyen műveleteket végeztünk, amikor a minimális távolságot megkaptuk. Ez lesz az Edit Distance algoritmus második lépése.

Szerencsére az algoritmus első lépése nemcsak a távolságot adta meg, hanem a mátrixot is. Ebből a mátrixból kiolvasható, hogy mikor melyik lépést alkalmaztuk *(törlés, beszúrás, egyezés, csere)*.

Induljunk el $M_{|s|,|t|}$-ből, és lépjünk visszafelé a mátrixban! Minden lépésben megnézzük, hogy az érték kiszámításakor melyik kifejezés szolgáltatta a minimumot. Ha törlés volt, balra lépünk, ha beszúrás, akkor felfelé, ha pedig csere vagy egyezés, akkor átlósan. Ha egyezés volt, akkor az aktuális indexeket eltároljuk. Ezek az indexpárok adják a kapcsolódási pontokat, vagy *link*eket. Az algoritmus akkor ér véget, ha eljutottunk $M_{0,0}$-ba.

Kicserélhető részmondatok

A két mondat szomszédos linkek közötti részmondatait kicserélhetőeknek tekintjük. További hipotézisként feltesszük, hogy ha két részmondat kicserélhető, akkor ők azonos szerepet töltenek be a két mondatban. Ez alapján a részmondatokat típusokba sorolhatjuk. Az eddigiek szerint tehát a kicserélhető részmondatok azonos típusúak. Az ABL algoritmus első felének célja, hogy meghatározza a részmondat-típusokat.

3.3.2. Típusok összefűzése

Ott tartunk tehát, hogy a részmondatokat kettesével típusokba soroltuk. Így most minden típusunk kételemű. Menet közben azonban kiderülhet, hogy két

részmondatpár azonos típusba sorolható. Amikor új párt találunk, az alábbi esetek fordulhatnak elő:

1. A két részmondat teljesen új, nem találtuk meg őket korábbi összehasonlítások alkalmával.
2. Az egyik részmondat már szerepel valamelyik típusban.
3. Mindkét részmondatot felfedeztük előzőleg.

Az első esetben semmi dolgunk nincs, el kell tárolnunk a részmondatokat egy új típusban, ahogy eddig is tettük. A második esetben azt a részmondatot, amelyikkel még nem találkoztunk, besoroljuk a már ismert részmondat típusába.

A harmadik eset kissé bonyolultabb: Ezen belül is két eset lehetséges. Ha mindkét részmondat ugyanabban a típusban van, nem kell tennünk semmit. Ha viszont különböző típusban vannak, az egyik típust meg kell szüntetnünk, elemeit pedig a másik típusba kell „költöztetnünk".

Kissé merész megközelítés, hogy típusokat egyetlen bizonyíték alapján összevonjunk. Főleg — ahogy azt később látni fogjuk — annak tudatában, hogy előfordulhat, hogy ezt a bizonyítékot az algoritmus második részében töröljük. Ezt a problémát Zaanen is megemlíti dolgozatában alternatív megoldásokat kínálva, ennek ellenére algoritmusa ezt a módszert használja. Dolgozatom egy későbbi fejezetében én is megpróbálok néhány javaslatot tenni a probléma kiküszöbölésére.

3.4. Selection Learning

Az algoritmus első részében találtunk rengeteg részmondatot. Most az a feladatunk, hogy kiszűrjük és töröljük ezek közül a „zavaróakat".

Azt mondjuk, hogy két részmondat *átfedő*, ha metszetük nemüres, de egyik sem része a másiknak, azaz:

$$x_1 < y_1 \leq x_2 < y_2,$$

ahol x_1 és x_2 az x részmodat első illetve utolsó szavának helye a mondatban, y-ra hasonlóan.

Alapvető hipotézisünk, hogy a tanulni kívánt nyelvet környezetfüggetlennek tekintjük. Ez a természetes nyelvekre nyilvánvalóan nem igaz. Azért élünk ezzel a feltevéssel, mert csak így van esélyünk arra, hogy használható szabályokat fedezzünk fel. Az előbbiekben feltárt kicserélhető részmondatok fogják alkotni szabályaink jobb, illetve bal oldalát. Ahhoz, hogy a nyelvtan környezetfüggetlen legyen, nem engedhetünk meg átfedő részmondatokat. Célunk minden mondatban részmondatoknak egy olyan átfedésmentes rendszerét találni, amely valamilyen értelemben optimális. Arra, hogy mit nevezünk optimálisnak Zaanen 5 különböző választ ad. Én ezek közül kettőt fogok részletesen leírni.

3.4.1. Részmondatok gyakorisága

Ahhoz, hogy kiválasszuk, mely részmondatok „fontosabbak", meg kell vizsgálnunk, hogy egy-egy részmondat milyen gyakran szerepel a különböző típusokban. Amikor típusokat akartunk összefűzni, egyenlőnek tekintettünk két részmondatot, ha azok ugyanabban a mondatban ugyanazon a helyen voltak. Ha most is így vizsgálnánk, minden részmondat különböző lenne, hiszen az azonosakat már összevontuk. Ehelyett most akkor nevezünk két részmondatot egyenlőnek, ha a *tartalmuk* azonos. Ezek alapján két lehetőség van arra, hogyan definiáljuk egy részmondat gyakoriságát:

- Megkeressük az összes azonos részmondatot, majd számukat elosztjuk az összes részmondat számával. (leaf)
- Az azonos részmondatokat csak a részmondat típusán belül keressük, végül a típus elemszámával osztunk. (branch)

Ez a két lehetőség fogja megadni a Selection Learning algoritmus két fajtáját.

3.4.2. Részmondat-halmazok „jósága"

Legyen $H = \{x_1, x_2, \ldots\}$ részmondatok egy halmaza. *H jóság*án a következő képlettel megadott számot értjük:

$$p_H = \sqrt[|H|]{\prod_{i=1}^{|H|} p_i}$$

ahol p_i az x_i részmondat gyakorisága. Egy részmondat-halmaz jósága tehát a részmondatok gyakoriságának mértani közepe.

Most, hogy definiáltuk egy részmondat-halmaz jóságát, kimondhatjuk, hogy keressük átfedésmentes részmondatoknak azt a maximális (nem bővíthető) halmazát, melynek jósága a legnagyobb. Ha a maximumot több ilyen halmaz jósága is felveszi, közülük azt választjuk, amelynek elemszáma a legnagyobb. Abban, hogy megtaláljuk ezt a halmazt, megint a dinamikus programozás lesz a segítségünkre.

3.4.3. Az algoritmus

Első lépésként ki kell számolnunk a részmondatok gyakoriságát. Az üres és az egyszavas részmondatok soha semmivel nem lesznek átfedőek, ezért minden átfedésmentes halmazba berakhatók. Ezeknek a gyakoriságára tehát nem is lesz szükségünk. Ha ezzel megvagyunk, indulhat a rekurzió. Az alapötlet ugyanaz, mint az Alignment Learning esetében: Ne foglalkozzunk rögtön az egész mondattal, csak bizonyos „csonkított" részeivel! Definiáljuk a $H_{i,j}$ halmazt minden $1 \leq i < j \leq |s|$ számpárra úgy, hogy $H_{i,j}$ annak a mondatnak az optimális halmaza, amely az eredeti mondatbol az i előtti és a j utáni rész levágásával keletkezik. A nagyobb intervallum optimumának elkészítéséhez szükségünk lesz a következő apró állításra:

1. Lemma. *Jelölje $M_{i,j}$ az i-edik szótól a j-edik szóig tartó mondatban az összes tartalmazásra nézve maximális részmondathalmazok rendszerét! Ekkor $\forall 0 \leq i < j \leq |s|$-re és $\forall H \in M_{i,j}$-re $\exists i \leq k < j, H_1 \in M_{i,k}, H_2 \in M_{k+1,j}$, melyekre $H = H_1 \cup H_2 \cup [i,j]$, ahol $[i,j]$ üreshalmaz, kivéve ha létezik az i-től j-ig tartó részmondat.*

Bizonyítás: Ha $H = [i,j]$, akkor az állítás triviális. Tegyük tehát fel, hogy $H - [i,j] \neq \emptyset$. Ekkor létezik $H - [i,j]$-ben egy leghosszabb részmondat (nem

feltétlenül egyértelműen). Legyen ez a részmondat $[k_1, k_2]$. Nem lehet, hogy $k_1 = i$ és $k_2 = j$. Az általánosság megsértése nélkül feltehetjük, hogy például $k_1 \neq i$. Azt állítjuk, hogy $H = H_1 \cup H_2 \cup [i, j]$, ahol $H_1 \in M_{i,k_1-1}, H_2 \in M_{k_1,j}$, H_1-et és H_2-t úgy kapjuk, hogy H-ból elhagyjuk azokat a részmondatokat, melyek nem illeszkednek a megfelelő intervallumba. Tegyük fel ugyanis, hogy nem így van, tehát hogy létezik H-ban olyan részmondat, melynek l, m határaira $l \leq k_1 - 1 < m$. Ha $m < k_2$, akkor ez a részmondat átfedésben van $[k_1, k_2]$-vel, ami nem lehetséges. Ha viszont $m \geq k_2$, akkor a vizsgált részmondat hosszabb, mint $[k_1, k_2]$, ellentétben a feltevéssel. A lemma igazolásához kell még, hogy a fenti módszerrel előállított H_1 (és H_2) nem bővíthető. Tegyük fel például, hogy létezik olyan részmondat, mellyel H_1 bővíthető. Mivel ez a részmondat biztosan nincs átfedésben H_2 elemeivel, ugyanezzel a részmondattal a H-t is bővíthettük volna.•

A lemma alapján világos, hogy

$$H_{i,j} = OPT_{k=i+1}^{j-1} \{(H_{i,k} \cup H_{k,j}) \cup [i, j] \,|(H_{i,k} \cup H_{k,j}) \cup [i, j] \, nem \; bővíthető\}$$

Az általános lépésben úgy határozhatjuk meg $H_{i,j}$-t, ha előzőleg az összes $j-i$-nél rövidebb „csonkamondatra” már meghatároztuk. Az egyetlen dolog, amire figyelnünk kell tehát, hogy az algoritmus ciklusváltozója a csonkamondatok méretét adja meg.

3.5. Összefoglalás

Foglaljuk össze, milyen lépéseket milyen sorrendben csinál az algoritmus!

- Vesszük az összes lehetséges mondatpárt, megkeressük bennük a kicserélhető részmondatokat.
- Közben ha olyan részmondatot találunk, ami már szerepelt korábban, elvégezzük a megfelelő típusösszevonás műveletet.
- Minden részmondatnak kiszámoljuk a gyakoriságát (kivéve az üres és az egyeleműeket)

- Mondatonként elvégezzük a Selection Learning műveletet: megkeressük az optimális részmondat-rendszert.

3.6. Eredmények

Ebben a részben Zaanen saját maga által publikált eredményeit írom le. Az én teszteredményeimet az 5. fejezetben fogom tárgyalni.

Ahhoz, hogy az ABL algoritmust teszteljük, két dologra van szükségünk:

1. Olyan korpuszra, melyet előzőleg nyelvészek elemeztek: az eredeti elemzéssel fogjuk összehasonlítani az algoritmus eredményét.

2. Egy mértékre, ami egy mondat kétféle elemzését összehasonlítja, majd százalékos arányban meg tudja adni, hogy a két elemzés mennyire hasonlít egymásra. (Ha a két elemzés teljesen azonos, akkor az eredmény 100%)

3.6.1. Korpuszok

Két különböző korpuszt használt Zaanen a teszteléshez. Az egyik az ATIS korpusz[8] (Air Travel Information System). Ez egy angol nyelvű szövegállomány, 716 mondatot és 11 777 megjelölt részmondatot tartalmaz. A korpuszban légiforgalmi irányítók beszélgetéseiből kiollózott részletek találhatóak. A másik az OVIS korpusz[9] (Openbaar Vervoer Informatie Systeem – Tömegközlekedési információs rendszer) 10 000 holland nyelvű, utazási információkat tartalmazó mondatból áll.

3.6.2. Mértékek

Az algoritmus hatékonyságának mérésére több különböző mérték is született, melyek nem teljesen ugyanazt a sorrendet állapítják meg a program különböző változatai között. Az alábbiakban definiálom a három mértéket:

- $NCBP = \frac{\sum_i |O_i| - |Cross(O_i, T_i)|}{\sum_i |O_i|}$
- $NCBR = \frac{\sum_i |T_i| - |Cross(T_i, O_i)|}{\sum_i |T_i|}$

- $ZCS = \frac{\sum_i Cross(O_i,T_i)=0}{|TEST|}$

A jelölések:

O_i és T_i az i-dik mondat részmondatainak halmaza az eredeti és az eredmény szerint.

$Cross\,(U,V)$ azon részmondatok az U-ból melyek legalább egy V-beli részmondattal átfedésben vannak.

$TEST$ Az algoritmus által kiadott összes részmondat.

A mértékek szemléletes jelentése:

- *Non-Crossing Brackets Precision*: Azon megtanult részmondatok aránya, melyek nincsenek átfedésben egyetlen korpuszbeli részmondattal sem.

- *Non-Crossing Brackets Recall*: Azon korpuszbeli részmondatok aránya, melyek nincsenek átfedésben egyetlen tanult részmondattal sem.

- *Zero-Crossing Sentences*: Azon *mondatok* aránya, melyekben egyáltalán nincs átfedő a tanult és az eredeti részmondatok között.

A most következő táblázat mutatja az algoritmus különböző változatait a két korpuszon mindhárom mérték szerint elemezve. A táblázatban lévő számok jelentése százalékban: „átlag (szórás)”.

	ATIS korpusz		
	NCBP	NCBR	ZCS
default leaf	82,31 (0,32)	83,10 (0,31)	22,02 (0,76)
default branch	86,04 (0,10)	87,10 (0,09)	29,01 (0,00)
biased leaf	81,43 (0,32)	83,11 (0,31)	22,44 (0,70)
biased branch	85,31 (0,10)	87,13 (0,09)	29,71 (0,00)
	OVIS korpusz		
	NCBP	NCBR	ZCS
default leaf	85,67 (0,02)	79,95 (0,03)	30,90 (0,08)
default branch	89,39 (0,00)	84,90 (0,00)	42,04 (0,02)
biased leaf	85,25 (0,02)	79,88 (0,03)	30,89 (0,08)
biased branch	89,25 (0,00)	85,04 (0,00)	42,19 (0,01)

Nézzük, mit mondanak ezek az adatok az algoritmus különböző változatai közti különbségről!

Az első, amit rögtön észreveszünk, hogy a Selection Learning fajtái között nagy a különbség. Az összes adat azt mutatja, hogy *branch* módban sokkal nagyobb hatásfokot érünk el, mint *leaf* módban. Az Alignment Learning módjainál már távolról sem ilyen egyértelmű a helyzet. Az esetek egy részében az derül ki, hogy a *default* módszer a jobb, máshol pedig a *biased* a nyertes. A legtöbb helyen viszont olyan kicsi az eltérés a két mód eredményei között, hogy hiba lenne ez alapján eldönteni, melyik a jobb. Az ilyen kicsi különbségek inkább vehetőek statisztikai hibának, mint komoly felfedezésnek.

4. fejezet

Problémafelvetések

Az ABL algoritmus felvet néhány problémát, melyekkel érdemes foglalkozni. Ezek egy részét Zaanen is említi dolgozatában, néhányukra megoldási lehetőségeket is felvet. Először ezekkel fogunk foglalkozni, a fejezet második felében pedig saját felvetéseimet írom le.

4.1. Típusösszevonások

Amint az eddig leírtakból is látszik, az algoritmus jelenlegi változatában két típust akkor vonunk össze, ha találunk olyan példát, ahol egy adott részmondat mindkét típusban szerepel. Ez a megoldás azért veszélyes, mert előfordulhat, hogy két szintaktikailag egészen más funkciót betöltő részmondat egyetlen példa alapján „találkozik", és innentől kezdve ők ugyanolyan típusúnak fognak számítani.

Ennek a hibának a kiküszöbölésére az alábbi megoldás kínálkozik: Változtassuk meg a típusösszevonás műveletet! Ne vonjunk össze két típust egyetlen „bizonyíték" alapján. Helyette számoljuk az eseteket, ami alapján összevonnánk, és ha elég bizonyíték gyűlik össze, akkor vonjunk össze. Ez a módszer biztosítja, hogy „véletlenül" ne vonjunk össze típusokat. Az viszont, hogy mikor tekintünk egy egyezést bizonyítottnak, újabb problémákat vet fel. Az is előfordulhat, hogy több egymásnak ellentmondó egyezésre is elég példánk van, ilyenkor el kell döntenünk, hogy melyik valószínűbb. Ezen a ponton kezd el igazán bonyolulttá válni a probléma. Vegyük a részmondatok halmazát és értelmezzünk ezen a halmazon egy távolságfüggvényt: Két részmondat legyen

egymáshoz annál közelebb, minél több esetben cserélhetjük ki őket egymással. Ebben a térben kellene valamiféle klaszterezési eljárást lefuttatni, és az egyes klaszterek lesznek a típusok. Az, hogy mi legyen ez az eljárás, akár egy teljes dolgozat témája is lehetne.

Az algoritmus típusösszevonási műveletével van még két probléma, melyek talán még súlyosabbak, mint az imént említett.

- Tegyük fel, ahogy az előbb, két típust összevontunk egy bizonyos kicserélhető részmondatpár felfedezésekor. Miután az Alignment Learning rész lefutott, minden típusösszevonás megtörtént, jön a Selection Learning. Előfordulhat, hogy ebben a részben azt a részmondatot, ami alapján csináltuk az összevonást, el fogjuk hagyni. Sőt, lehet, hogy mindkét mondatban elhagyjuk a megfelelő részmondatot. Mostmár duplán hibáztunk: Nemcsak hogy egyetlen bizonyíték alapján vontunk össze két típust, de még ki is töröltük később a bizonyítékot! A törlés akkor is okozhat ilyen problémát, ha az összevonásra sok bizonyíték volt: lehet, hogy az összes bizonyítékot töröljük később.

- Kövessük most végig egyetlen típus útját az algoritmusban! Az első részben néhányszor összevontuk vagy bővítettük, mondjuk l elemű lett az Alignment Learning végére. Ezek után nézzük mi történik vele a második részben! A típus néhány eleme törlődik, a többiek megmaradnak. Mi van akkor, ha pont $l - 1$ elemet törlünk, és 1 elemet hagyunk meg? Az algoritmus kimenetében lesz egy típus, aminek csak egy eleme van! Ez a típus semmilyen információt nem hordoz, viszont ez az egy elem a neki megfelelő mondatban kiszoríthatott egy fontosabb részmondatot.

Meglehetősen furcsa, hogy ez a két elvi hiba bennemaradt az algoritmusban, annak ellenére, hogy igen egyszerű őket kiküszöbölni: változtassunk egy kicsit az algoritmus lépéseinek sorrendjén! A típusösszevonást hagyjuk a Selection Learning után! Ha előbb szűrjük ki a szükségtelen részmondatokat, és csak utána vonunk össze, akkor a fenti két hiba egyike sem fordulhat elő.

4.2. Futási idő

Az algoritmus első részeben (Alignment Learning) minden mondatpárra lefuttatjuk az edit distance eljárást. Ebből következik, hogy futási ideje a mondatok számában négyzetes. Ez pár ezer mondatból álló korpusznál még nem okoz problémát. Ha viszont használható eredményhez akarunk jutni, ennél nagyságrendekkel nagyobb korpuszon kell futtatnunk a programot. Ha az input több százezer mondatból áll, tapasztalatom szerint egy átlagos személyi számítógépen hetekig is eltarthat, mire végigfut a program. Ahhoz, hogy gyorsabbá tegyük az algoritmust, el kell kerülnünk, hogy az összes mondatpárt megvizsgáljuk. De hogy döntjük el, hogy mely mondatokat melyekkel hasonlítsunk össze? Erre megint több lehetőség adódik.

Mindenekelőtt el kell döntenünk, hogy egy-egy mondatot hány másikkal párosítsunk össze. Ha ezt a számot növeljük, nő a futási idő, ugyanakkor – több információ lévén – javítja az algoritmus hatásfokát. De bármilyen értéket adjunk is meg, a futási idő így már lineáris lesz.

- Első megközelítésben konstruáljunk egy véletlen gráfot, melynek csúcsai a mondatoknak felelnek meg, ha pedig két csúcs között megy él, akkor a megfelelő mondatokat össze fogjuk hasonlítani. Az aligment learning részben az élek mentén fogjuk a mondatpárokat elemezni. Szükségünk lenne tehát egy véletlenül generált egyszerű (párhuzamos és hurokélmentes) reguláris gráfra, ahol minden csúcs foka egy előre megadott k szám. Ilyen gráf előállítása nem könnyű feladat (oszthatósági feltételek miatt nem is mindig lehetséges). Helyette nekünk elég, ha olyan véletlen gráfot generálnunk, amely „majdnem” reguláris, a fokszámok átlaga pedig k. Ilyen gráfot rendkívül egyszerűen elő tudunk állítani. Az algoritmus további működése azonos a fentebb leírtakkal.

- Az első módszerrel az a baj, hogy nagyon sok olyan mondatpár van, melyekben alig, vagy egyáltalán nincs közös szó. Ezekből a mondatpárokból nemigen tudunk értékes információt kinyerni. Ha a korpuszban „vakon” keressük meg azokat a mondatpárokat, melyeket össze akarunk hasonlítani, akkor félő, hogy elkerüljük azokat a párosításokat, melyekből a legtöbbet tudhatunk meg. így sok munkával juthatunk felesleges

információkhoz. (Két teljesen különböző mondat elemzésénél például csak azt tudhatjuk meg, hogy az egyik mondat teljes egészében lecserélhető a másikra.) Jobb módszer lenne tehát, ha a gráfot nem teljesen véletlenül generálnánk. Ehhez szükség van egy előzetes elemzésre, mely megtippeli, hogy az egyes mondatpárok milyen mértékben hasonlítanak egymásra (pl. közös szavak számának kiszámításával). Ha ez megvan, akkor elő tudjuk állítani a véletlen gráfot úgy, hogy ha két mondat között nagyobb a hasonlóság, akkor a nekik megfelelő él nagyobb valószínűséggel kerül bele a gráf élhalmazába.

4.2.1. Megvalósítás

A második módszernél felmerül a kérdés, hogyan lehet az előzetes elemzést elkészíteni úgy, hogy ne legyen négyzetes az algoritmus. Erre a problémára egy megoldás a következő: Menjünk végig a mondatokon, és minden szónál jegyezzük fel, hogy az aktuális mondatban az aktuális szó szerepel. Így végül kapunk minden szóra egy listát azokról a mondatokról, melyekben az adott szó szerepel. Két mondat akkor tekinthető hasonlónak, ha viszonylag sok olyan szó van, ami mindkettőben benne van. (Most a hasonlóságnak ez a gyengébb definíciója is elég nekünk.) Egy adott mondathoz hasonló mondatokat úgy találunk, hogy a mondatban lévő szavakról lekérdezzük, hogy mely más mondatokban szerepelnek. Így minden mondathoz kaphatunk egy listát a lehetséges (gráfbeli) szomszédairól. Sőt, a közös szavak számának ismeretében még rangsorolhatjuk is a jelölteket. Innentől a véletlen gráf előállítása semmivel nem nehezebb, mint az első módszernél. Láthatjuk azt is, hogy ez a „szomszédjelölt-válogatás” lineáris időben lefut, hiszen egyszer megyünk végig a mondatokon, ezen belül nincs más ciklus.

Következő lehetőségként adódik, hogy ha már rangsoroltuk a mondatokat aszerint, hogy hány közös szavuk van az aktuális mondattal, akkor nem is érdemes a gráf éleit sorsolással választani. Javíthatja az algoritmus hatásfokát, ha véletlen helyett kiválasztjuk a rangsorolásban az első k mondatot, és azok lesznek az aktuális mondattal összehasonlítandóak.

Érdekes új kérdéseket vet fel ez a módosítás. Mennyi legyen az a bizonyos k érték? Ennek megválaszolásához el kell döntenünk, hogy mennyi időt

szánunk az algoritmus futtatására. Egy másik szempont is felmerül: érdemes megvizsgálni, hogy meddig növelhető k úgy, hogy a végeredményben jelentős különbséget okozzon? Lehetséges-e, hogy egy idő után már hiába párosítok több mondatot, a kicserélhető részmondatok száma és gyakorisága nem változik jelentősen? Ha igen, hogyan lehet megtalálni az optimális átlagfokszámot?

4.3. Általánosított részmondatok

Emlékeztetőül: részmondatnak neveztük azt a szósorozatot, amely egy bizonyos mondatban egy adott szótól egy másikig tart. Ez a definíció (a kicserélhetőség elvével együtt) sok problémát okozhat. Azoknál a nyelveknél, ahol gyakori a ragozás és az egyeztetés, gondot okozhat, ha például a két egyeztetett szó közé beékelődik egy (vagy több) szó. Pl.:

I really love You.
He really loves You.

Az algoritmus sajnos azt fogja kimutatni, hogy az *I* kicserélhető *He*-re, a *love* pedig *loves*-ra, méghozzá egymástól függetlenül. Ez persze nyelvtanilag helytelen mondatokat eredményez. A magyar nyelvben még gyakoribbak az ilyen típusú mondatok. Érezhető, hogy a megoldás az lenne, ha az *I ... love* szerkezetet egyetlen részmondatnak tekintenénk. Így csak egyben tudnánk kicserélni a *He ... loves* szószerkezettel. Ehhez meg kell változtatnunk a részmondat definícióját: egy részmondat nem feltétlenül egy összefüggő szósorozatból áll, hanem több szósorozat egyesítése is lehet. Ettől persze nagyon elbonyolódik a kicserélhető részmondatok keresése, sőt, az átfedés definíciója sem világos. Elképzelhető, hogy ezzel az új definícióval teljes egészében át kellene írni az algoritmust.

4.4. Részmondatok súlyozása

A Selection Learning lépésben kétféleképpen is kiszámoltuk a részmondatok fontossági sorrendjét. Mindkét módszerben egyetlen adat játszotta a főszerepet: a részmondatok gyakorisága. Nem biztos, hogy ez az egyedüli szempont,

ami szerint rangsorolni lehet az egyes részmondatokat az optimális átfedésmentes halmaz megkereséséhez. Pár bekezdéssel korábban szó volt arról, hogy a különböző mondatpárok eltérő fontosságú információkat tartalmaznak. A részmondatok „selejtezésénél" azonban nem vesszük figyelembe, hogy egy-egy részmondat mennyire hasonló mondatokból keletkezett. Javaslatom szerint minden részmondathoz hozzá kéne rendelni egy súlyt, ami annál nagyobb, minél közelebb van egymáshoz az a két mondat, melyek vizsgálatánál a részmondatot megtaláltuk. Ezek után a gyakoriság és a súly megfelelő kombinációjával finomítani lehetne a Selecton Learning műveletet.

4.5. Iteráció

A legtöbb nyelvtanuló algoritmus fő fegyvere az iteráció: egy bizonyos műveletet végrehajt a korpuszon, majd a megváltozott korpusszal újra, és így tovább. Ezt lehet meghatározott lépésig csinálni, vagy addig, míg a korpusz el nem ér egy bizonyos szempontból stabil állapotot.

Az ADIOS algoritmus például addig ismétli a lépéseket, amíg már nem talál új szignifikáns mintát az aktuális gráfban. Az ABL algoritmus is alkalmas lehet egy ilyen módosításra. Ha a megtalált részmondat-típusoknak megfeleltetnénk egy-egy új szót (egy *nemterminálist*), módosíthatnánk a korpuszt úgy, hogy a régi mondatokban a kifejezéseket ezekkel az új szimbólumokkal helyettesítjük. Ezzel a módszerrel hatékonyabban találhatnánk meg az egymásba ágyazott részmondatokat, és egy lépésben csak diszjunkt részmondatokat tartanánk meg a Selection Learning alatt.

5. fejezet

Tesztek, eredmények

A fenti módosítások és ötletek közül néhányat (és persze az eredeti változatot) tesztelés céljából leprogramoztam. Ebben a fejezetben ezekről a programváltozatokról fogom megírni eredményeimet, tapasztalataimat. Két különböző tesztkorpuszt használtam: az egyik egy kb. 800 000 mondatból álló dokumentum, angol óvodások beszélgetéseiből (CHILDES – Child Language Data Exchange System). A másik az ATIS korpusz része, mely a Wall Street Journal című újságból válogatott mondatokból áll (WSJ). Általában a futtatáshoz egy párezres szeletet használtam. Ezen már látszik, hogy működik-e és hogy gyorsult-e a program, illetve hogy javult-e a hatásfok. A két korpusz között az egyik lényeges különbség, hogy míg a CHILDES-ben pár szavas mondatok vannak, addig a WSJ-ben nem ritkák a 100 szó körüli mondatok sem. Ezért a WSJ-n sokkal lassabban futott a program, mivel az algoritmus a mondatok hosszában $O\left(n^3\right)$-ös futási idejű.

A WSJ korpusz előnye, hogy rendelkezésemre áll egy *tagolt* változat, melyben a kifejezés-határok zárójelekkel be vannak jelölve. Mivel az ABL algoritmus egymásba ágyazott zárójeleket is talál, a tagolt korpuszban pedig nincsenek ilyenek, nem azt érdemes vizsgálni, hogy egy mondat pont úgy van-e elemezve, mint a program kimenete által, elég ha az eredmény zárójelei közt megtalálhatóak az eredeti zárójelek.

5.1. Az algoritmus gyorsítása

Több példányban lefuttattam a programot különböző paraméterekkel. Az összehasonlítási módszerem a következő volt:

- Minden outputot összehasonlítottam az eredeti tagolt korpusszal, és megvizsgáltam, hogy az eredeti korpusz zárójelei közül hányat talált meg a program, és hogy hány olyan mondat van, melynek az összes zárójelét megtalálta.
- Egymással összehasonlítottam a kimeneteket: megvizsgáltam minden mondatra, hogy a két kimenet által adott zárójelhalmazok metszetének elemszáma hányadrésze a két eredeti halmaz elemszámának atlagának.

Először a WSJ 23830 mondatos részén futtattam, abból néhány eredmény (a *mondatok száma* oszlopban az egy mondattal összehasonlítandó mondatok száma látható):

Alignment Learning	Selection Learning	Mondatok száma	Megtalált zárójelek(%)	Jó mondatok száma
default	branch	300	48,35	3683
		500	48,43	3676
		700	48,44	3691
		900	48,53	3690
		1000	48,63	3671
		1100	48,65	3665
		1300	48,68	3704
		1500	48,62	3677
		1700	48,72	3719
		1900	48,74	3731
		2100	48,75	3730
		2300	48,75	3747
		max.	47,90	3468
biased	branch	300	42,96	3625
		500	43,99	3651
		700	44,63	3707
		900	44,91	3707
		1000	44,93	3710
		1100	45,06	3712
		1300	45,17	3705
		1500	45,29	3619
		1700	45,42	3608
		1900	45,49	3603
		2100	45,57	3591
		2300	45,69	3592
		max.	45,90	3452

A következő táblázatokban az első és második módszer szerinti összehasonlítás eredményeit mutatom be. A korpusz a WSJ 2455 mondatos kezdőszelete volt.

Alignment Learning	Selection Learning	Mondatok száma	Megtalált zárójelek(%)	Jó mondatok száma
default	leaf	300	53,49	455
		500	53,80	454
		700	53,61	467
		900	53,91	468
		1000	53,93	471
		1100	54,08	465
		max.	53,87	458
default	branch	300	48,94	396
		500	49,24	402
		700	49,70	423
		900	49,69	413
		1000	49,49	404
		1100	49,45	399
		max.	49,23	390
biased	leaf	300	44,06	414
		500	45,38	414
		700	45,62	404
		900	45,72	405
		1000	45,73	407
		1100	45,59	409
		max.	45,65	410
biased	branch	300	41,71	386
		500	43,09	385
		700	43,18	392
		900	43,20	387
		1000	43,31	384
		1100	43,29	385
		max.	43,25	374

A második táblázatban a lineáris algoritmusok eredményét hasonlítom össze az eredeti algoritmus eredményével.

		Átlag	Szórásnégyzet	Egyező mondatok
Def. Leaf	300	0,809	0,0145	326
	500	0,853	0,0134	509
	700	0,704	0,0202	182
	900	0,734	0,0138	195
	1000	0,754	0,0186	253
	1100	0,765	0,0184	269
Def. Branch	300	0,794	0,0182	271
	500	0,838	0,0171	418
	700	0,872	0,0158	648
	900	0,899	0,0137	878
	1000	0,912	0,0127	1018
	1100	0,920	0,0123	1151
Bia. Leaf	300	0,807	0,0187	405
	500	0,867	0,0154	700
	700	0,907	0,0126	1036
	900	0,936	0,0098	1368
	1000	0,947	0,0085	1534
	1100	0,955	0,0073	1669
Bia. Branch	300	0,783	0,0239	337
	500	0,844	0,0214	575
	700	0,881	0,0187	840
	900	0,913	0,0156	1181
	1000	0,925	0,0142	1336
	1100	0,935	0,0125	1471

Az összes adat azt mutatja, hogy 900 fölé nem nagyon érdemes emelni az adott mondattal összehasonlított mondatok számát. (Ez csak a 2455 mondatos változatra érvényes, a 23830 mondatos korpuszon ez az optimum magasabb.) Sőt, az is kiderült, hogy 900 felett nemcsak nem nő tovább a hatékonyság, hanem meglepő módon csökken. Még az eredeti (minden mondatot minden mondattal összehasonlító) algoritmus is rosszabb, mint a 900 körüli

átlagfokszámú. Bevallom, erre nem számítottam, amikor a program gyorsítására irányuló változtatást terveztem.

Utólag megvizsgálva megmagyarázható ez a jelenség. Mivel a mondatokat a hasonlóság sorrendjében vizsgálom, minél nagyobb a paraméter, annál kevésbé hasonló mondatok is összehasonlításra kerülnek. Az ezekből az összehasonlításokból származó részmondatok esetenként kiszoríthatnak más, esetleg több információt hordozó részmondatot. Így félrevihetik az elemzést, minek következtében kevesebb helyes zárójel kerül be a végeredménybe.

Az algoritmus váratlan javulásából arra következtettem, hogy még hatékonyabbá lehetne tenni a programot, ha a paraméterrel nem az átlagfokszámot adnám meg, hanem az összehasonlítandó mondatok közös szavainak minimális számát. Így biztosak lehetnénk benne, hogy csak olyan mondatokat hasonlítottunk össze, melyek érdemben segítik az elemzést. Ehhez meg kéne vizsgálni, hogy mi az optimális paraméter, amivel a megtalált zárójelek számát maximalizálni lehet. Az is elképzelhető, hogy a közös szavak számának minimumát nem konstans paraméterrel érdemes beállítani, hanem egy a mondat hosszától függő függvényt kéne előállítani.

Mivel mindkét ötlet további elemzéseket, kutatást igényel, ezeket jelen keretek közt nem áll módomban kifejteni.

5.2. A lépések megfordítása

Amikor hozzáfogtam, hogy elkészítsem az algoritmus módosított lépéssorrendű változatát, egyre-másra merültek fel az újabb kérdések, problémák, pl.:

- Ha egy típus egyik tagját (most minden típusnak pontosan két tagja van) nem használtam a megfelelő mondatban, akkor be kell tiltanom a másikat.

- Ha a típus egyik tagját felhasználtam, a másik tagját pedig *később* nem, akkor mit csináljak a már felhasznált részmondattal? Ha visszamenőleg betiltom, akkor azon a mondaton futtassam újra a Selection Learninget, vagy hagyjam meg foghíjasan a részmondatrendszert?

- Mivel a részmondatok gyakoriságát még a típusösszevonás előtt kell megcsinálni, a legtöbb részmondat nagyon sok példányban lesz jelen a memóriában. Ez egyrészt elfogyasztja a memóriát, másrészt (ami sokkal nagyobb gond), mérhetetlenül lelassítja a program futását.

Végül a következő megvalósítást választottam:

- Az Alignment Learning részben párhuzamosan két példányban tárolom el a megtalált részmondatokat: az egyiknél megcsinálom az összevonást, a másiknál minden típus kételemű marad. Az összevont adatbázist csak arra használom, hogy kiszámítsam a részmondat-gyakoriságokat, utána ki is törlöm.

- A selection Learning részben, ha egy mondatban egy részmondatot nem választ ki az algoritmus, akkor az annak a részmondatnak megfelelő *összes* típust törlöm, illetve letiltom. Ez azt jelenti, hogy a korábban vizsgált mondatokból kihúzom a részmondatot, a később vizsgáltaknál pedig nem veszem figyelembe. Így a régebbi mondatokban a megtalált részmondat-rendszer nem lesz maximális. Ha ugyanis megpróbálnám újra elemezni a „sérült" mondatokat, akkor ott újabb részmondatokat kéne betiltani, sőt, lehet, hogy betiltott részmondatokat kéne felélesztteni. Ezzel akár végtelen ciklusba is kerülhet a program, de mindenképpen sokkal lassabb lesz.

- Az összevonás műveletnél a részmondatokat a megtalálás sorrendjében veszem (mintha most lennénk az Alignment Learningnél), és így vonom össze őket. A betiltott típusokat természetesen kihagyom.

Az a megoldás, hogy a típusokat két példányban tárolom elég helyigényes, ami sajnos átlagos memóriamérettel rendelkező gépeken jelentős lassulást okoz. De még így is gyorsabban fut le a program, mintha az összevonás előtt kéne a részmondat-gyakoriságokat kiszámolni.

Amint azt a lenti táblázat is mutatja, ez a módosítás nem váltotta be a hozzá fűzött reményeket. Nem elég, hogy lassabb és nagyobb tárigényű lett a program, sajnos az eredmények sem javultak (legalábbis a WSJ korpuszon). Az új algoritmus minden általam mért ellenőrző paramétere rosszabb, mint

az eredetié. Lehet, hogy a részmondat-betiltás túl drasztikus módszer, de az is lehet, hogy mégiscsak újra kellett volna futtatni az elrontott mondatokon a Selection Learninget. Elképzelhető, hogy az eredeti algoritmus pont ezek miatt a problémák miatt született meg mégis az eredeti műveleti sorrendben.

Alignment Learning	Mondatok száma	Megtalált zárójelek(%)	Jó mondatok száma
default	900	43,69	366
	1000	43,69	357
	1100	43,67	361
	max.	42,59	344
biased	700	39,06	339
	900	39,18	337
	1000	39,13	335
	1100	39,13	332
	max.	39,00	334

6. fejezet

Összefoglalás

Szakdolgozatomban megpróbáltam betekintést nyújtani az önszervező nyelvtanulás világába. Néhány algoritmus rövid bemutatása mellett részletes kritikát és elemzést adtam egy konkrét algoritmus, Menno M. van Zaanen 2001-ben publikált ABL algoritmusára. Megvizsgáltam, hogy milyen motivációk hatására alakult ki ez a tudományág, hogy hol tart, és hogy merre látok fejlődési lehetőséget. Az ABL algoritmus elemzésénél több konkrét javaslatot tettem a hatékonyság javítására. Ezek közül kettőt ki is próbáltam. Az egyikről kiderült, hogy jelenlegi formájában inkább ront a programon. A másik újítás, ami eredetileg csak az algoritmus felgyorsítását célozta, végülis azt a meglepő eredményt hozta, hogy nemcsak jóval gyorsabbá, de hatékonyabbá is tette az ABL algoritmust. Ezt az állításomat hosszas tesztelés eredményei is igazolják.

Köszönetnyilvánítás

Ezúton szeretném megköszönni mindazoknak, akik segítettek abban, hogy ez a szakdolgozat megszülessen. Köszönöm a szakmai segítségnyújtást Szamonek Zoltán kollégámnak és Szepesvári Csaba témavezetőmnek, hogy irányt mutattak, javaslataikkal segítettek. Köszönöm családomnak, legfőképp menyasszonyomnak, Pákozdi Ágnesnek, hogy lelki támogatást nyújtottak szakdolgozatom írása közben és előtt.

Irodalomjegyzék

[1] Menno M. van Zaanen: *Bootstrapping Structure into Language: Alignment-Based Learning*, arXiv:cs.LG/0205025 v1

[2] Ido Dagan, Lillian Lee, Fernando C. N. Pereira: *Similarity-Based Models of Word Cooccurrence Probabilities*, 1999. Kluwer Academic Publishers, Boston

[3] Kornai András: *How many words are there?* Glottometrics 4, 2002, 61-86.

[4] Slava M. Katz: *Estimation of probabilities from sparse data for the langauge model component of a speech recognizer.*, 1987. IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-35(3):400–401, March.

[5] Fernando Pereira, Naftali Tishby, Lillian Lee: *Distributional Clustering of English Words* 1994. arXiv: cmp-lg/9408011 v1

[6] Zach Solan, David Horn, Eytan Ruppin, Shimon Edelman: *Unsupervised learnig of natural languages*

[7] Delphine Bernhard: *Unsupervised Morphological Segmentation Based on Segment Predictability and Word Segments Alignment*

[8] ATIS corpus, Marcus et al., 1993.

[9] OVIS corpus, Bonnema et al., 1997.