

Szegedi Tudományegyetem
Informatikai Intézet

SZAKDOLGOZAT

Gyúróczki Gergő Viktor
2025

**Szegedi Tudományegyetem
Informatikai Intézet**

**Magyar szövegek adatbázisának kialakítása,
vizsgálata**

Szakdolgozat

Készítette:

Gyúróczki Gergő Viktor
programtervező informatika szakos
hallgató

Témavezető:

Dr. Csirik János
professor emeritus

Szeged
2025

Feladatkiírás

Jelentős számban elérhetőek magyar nyelven irodalmi művek az interneten. A cél ezek adatbázisba foglalása és alapvető nyelvészeti és statisztikai vizsgálatok elvégzése.

Tartalmi összefoglaló

A téma megnevezése:

Magyar szövegek adatbázisának kialakítása, vizsgálata

A megadott feladat megfogalmazása:

Könyvekből szöveg kinyerése, kinyert szöveg elemzése

A megoldási mód:

Szövegkinyerés a könyvek .pdf fájljaiból OCR segítségével, a kimenet szkriptekkel való tisztítása és formázása az elemezhetőség javítása érdekében. A kimenet elemzése magyarlánccal és CoreNLP-vel, ezen kimenetek vizualizációja szófelhőkkel és a kimeneti adatok összehasonlítása más releváns adatokkal diagramokon keresztül. A kimenetben fellelhető hibák és észrevételek összegyűjtése.

Alkalmazott eszközök, módszerek:

- magyarlanc 3.0
- CoreNLP 4.4.0
- Tesseract
- Matplotlib
- Eclipse IDE
- Visual Studio Code

Elért eredmények:

Munkám eredményeként megismerkedhettem a szövegkinyerés és nyelvi elemzés problémáival, a magyarlanc működésével és feladataival. Elkészült egy projekt, amely több részből épül fel: szövegkinyerő, tisztító, elemző. Az elemzések eredményeinek vizualizálására megismerkedhettem különböző módszerekkel: diagrammok, szófelhők.

Kulcsszavak:

java, f#, python, magyarlanc, CoreNLP, OCR, elemzés, feldolgozás

Tartalomjegyzék

Feladatkiírás	3
Tartalmi összefoglaló	4
Tartalomjegyzék	5
Bevezető	6
1. Adatforrás	8
1.1 Szöveges adatforrás	8
1.2 MNSZ	10
2. Magyarlánc	11
2.1. Sentence Splitter & Tokenizer	12
2.2. Morphological analyzer	12
2.3. POS Tagger	12
2.4 Dependency parser	13
2.5 Használat	13
2.6 Bemenet	14
2.7 Kimenet	14
3. Magyarlánc alternatíva	14
3.1 Felépítés	15
3.2 Használat	15
3.3 Kimenet	15
3.4 Konklúzió	16
4. Projekt	16
4.1 Szövegkinyerés	17
4.2 Előfeldolgozás	19
4.3 Nyelvi elemzés	20
4.4 Utóelemző	23
5. Eredmények	27
5.1: Szófaji eloszlások	27
5.2: Szavak hossza és számuk	30
5.3: Szófelhők, szógyakoriságok	31
6. Összegzés	34
Irodalomjegyzék	35
Nyilatkozat	36
Köszönetnyilvánítás	36
Elektronikus melléklet	37

Bevezető

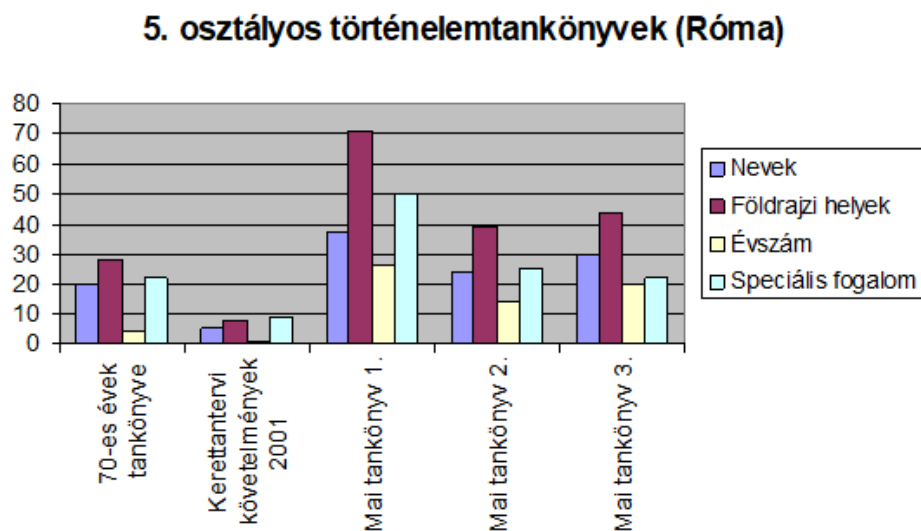
Az oktatásban használt tankönyvek tudatos elemzése, az elemzési paraméterek felügyelése és karbantartása nagyon fontos és érdekes téma, mivel ezekből a könyvekből tanulnak az újabb és újabb generációk. Mindenki érdeke a tankönyvek minőségének javítása, ugyanis a tanárok és diákok munkáját is megkönnyítik egyaránt. Vannak bizonyos paraméterek, amikre érdemes az idő múlásával követni és javítgatni azzal a céllal, hogy minél jobb minőségű könyveket lehessen előállítani [1]. Szinte minden témához találhatóak adatok: elvárt ismert szókincs ismerete korosztályonként, szakszavak, szófaji és szógyakorisági adatok, illetve tulajdonnevek gyakoriságának eloszlása.

Szerző	Életkor	Becsült szókincs
Crystal (1998) Gósy (1999)	3 év	1000–1200 A legkevesebb 150, a legtöbb 2500 (Gósy 1999: 189)
Crystal (1998)	4 év	1500
Crystal (1998)	5 év	2000
Crystal (1998) Büky (1984) Gósy-Kovács (2001)	6 év	2500 2000–3000 produktív Közül adatokat 8000–24 000 becslésről is.
Crystal (1998)	7 év	3000
Meixner (1971)	8 év	260–1468 produktív
Crystal (1998)	9 év	4000
Crystal (1998) Piéh (2006)	11 év 10 év felettiek	5000 30 000–40 000 az olvasottaké

Bev. 1 kép: ‘A szókincs becsült nagyságának alakulása az életkor előrehaladtával’

Az alábbi táblázatot nézve például egyértelműen látszik az, hogy az életkor növekedésével egyre jobban nő a becsült szókincs, ami teljesen logikus. Minden korosztálynál fontos arra figyelni, hogy olyan szókincset tartalmazó könyvek jussanak el hozzájuk, amiket van is esélyük megérteni. Ez mellett érdemes figyelni pl. arra is, hogy a mondatok szószáma ne legyen túl sok, azaz ne legyenek túl hosszúak a mondatok. Nagyon sokan kritizálják a mai tankönyveket, ugyanis véleményük szerint egyre jobban ‘szakkönyvek mint tankönyvek’, azaz egyre jobban elveszik a tankönyvek tanító jellege, és egyre jobban olyan érzést kelt a diákokban, mintha egy lélektelen dokumentációt olvasnának.

Ennek egy teljesen logikus oka lehet pl. a túl nagy szókincs, fogalmak, évszámok tudásának elvárása a diákoktól. A következő diagram az ezzel a témával foglalkozó ‘Korszerű, használható tankönyveket az iskolákba’ [2] című diasorból lett kiemelve:



Bev. 2 kép: 5. osztályos történelem tankönyvek statisztikái

A diagramon látszik, hogy az idő múlásával hogyan változott az ismerendő nevek, évszámok és helyszínek mennyisége. A 70-es évek könyvei jóval kevesebb tulajdonnév tudását követelték meg, mint a maiak. Az 5. osztály tapasztalataim szerint egy ‘választóvonal’ időszak, ugyanis ez volt az első évfolyam ahol sok olyan tárgy került be az órarendünkbe, ahol hirtelen sok tulajdonnév és fogalom ismerete lett a fontos, itt jelentkezett először az ún. ‘magolás’-ra alapozott számonkérés jelenléte.

Szakedolgozatom további részében a középiskolás 9-12. osztály irodalomkönyveinek az elemzéseiről és ezeknek az eredményeknek a kinyeréséhez elvezető útról lesz szó. Eredményeim közé tartozik majd az említett tankönyvek szövegeinek kinyerése, formázása, elemzése 2 különböző elemzőeszköz segítségével, majd ezeknek a kimenetének értelmezése és bizonyos kritériumonkénti lebontása, majd azok vizualizációja.

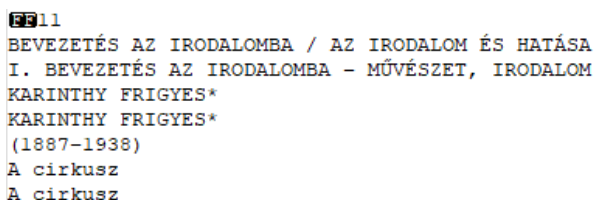
1. Adatforrás

1.1 Szöveges adatforrás

Kezdeti adatforrásként az Oktatási Hivatal által kiadott 9-12-es Irodalom tankönyvek és szöveggyűjtemények szolgáltak (ez 8 db könyvet jelent). Ezek hivatalos kiadásban elérhetőek a tankonyvkatalogus.hu [3] oldalon, ahol PDF formátumban külön-külön megtalálhatóak a könyvek. Egyéb metaadat nem állt rendelkezésre a végkimenet előállításához.

A könyvek PDF fájljai nem biztosítanak teljesen megbízható adatforrást, ugyanis a belőlük történő különböző szövegkinyerési módszerek alkalmazásával szavak kimaradhatnak, vagy bizonyos karakterek nehezen és hibával ismerhetőek csak fel. Egy ilyen kinyerési folyamat az OCR, amely a minőség függvényében viszonylagosan hosszú időt és számítási erőforrást igényel. Természetesen a legjobb eredményt úgy lehetett volna elérni, ha a tankönyvek kiadójától sikerült volna megszerezni az eredeti szövegek forrását, amire történt is kísérlet még az OCR-el történő szövegkinyerési gondolatok előtt. Ez azonban sikertelen volt, ugyanis az eredeti források nem találhatóak meg a kiadó elmondása szerint.

Kezdetekben történtek különböző kísérletezések online konvertálók [4] használatára, ám mindegyiknél az volt a tapasztalat, hogy a kinyert szöveg nagyon rossz minőségű és főképp hiányos volt, azaz nagy mennyiségű felhasználói beavatkozással sem lehetett volna megfelelő minőségű kinyerést véghez vinni velük. Ez mellett még a konfigurálhatósági lehetőségek sem voltak ideálisak, mint pl. az oszlopokba rendezett szöveg helyes kinyerésének konfigurálása, vagy a margókon levő szövegek levágása nem tűnt megvalósíthatónak. Ezeknek a hibáknak a mennyiségéről és a hibák konkrét típusainak számáról nem történt statisztikai elemzés, ugyanis bizonyos könyvek szövegkinyert változata szinte használhatatlan volt.



BEVEZETÉS AZ IRODALOMBA / AZ IRODALOM ÉS HATÁSA
I. BEVEZETÉS AZ IRODALOMBA - MŰVÉSZET, IRODALOM
KARINTHY FRIGYES*
KARINTHY FRIGYES*
(1887-1938)
A cirkusz
A cirkusz

1.1.1 kép: Online konvertálás módszerrel történt első oldal elejéből részlet


```
OH_IR9SZGY.indb 11
OH_IR9SZGY.indb 11 2021. 08. 27. 7:49:21
2021. 08. 27. 7:49:21
12
BEVEZETÉS AZ IRODALOMBA – MŰVÉSZET, IRODALOM
futott a lépcsőn. Aztán bársony tapétás szobákon
```

1.1.2 kép: Online konvertálós módszerrel kinyerés rossz kimenete

Ezek a képek is látszik, hogy szükség volt valami más módszerre. A fenti képen duplikált szöveg és hibás karakterek szerepelnek, az alsó képen pedig egy már ismétlődő cím sora, illetve egy mondat vége ami rossz helyre került. Ezek a hibatípusok különböző konverterek között eltért, de pl. az oszlopos formátumtól kinyerés és a több oldalra tagolt mondat hibák nagyon gyakoriak voltak.

A végső kinyerési módszer egy saját program lett, amely a Tesseract [5] nevű Java könyvtár használatával valósult meg. Mivel ez saját program írását követelte, a konfigurálhatóság szinte magától megoldódott. Helyes konfigurálás után jóval kevesebb hibával és felhasználói beavatkozással dolgozott, ám a kinyerés ideje sokszorosára nőtt. A kimenet tanulmányozása után látszott, hogy a kimenet sokkal jobb minőségű, ezért úgy döntöttünk, hogy a hosszabb feldolgozási idő ebben az esetben figyelmen kívül hagyható (mivel saját programról beszélhettünk, így természetesen a feldolgozási idő javítása is a mi kezünkben volt).

```
I. BEVEZETÉS AZ IRODALOMBA – MŰVÉSZET, IRODALOM

KARINTHY FRIGYES"
(1887–1938)

A cirkusz
```

1.1.3 kép: Saját programmal kinyert első oldal elejéből részlet

```
is látunk majd, de nem, széles, sok lépcső következett. Alig tudtam követni, oly sebesen
futott a lépcsőn. Aztán bársony tapétás szobákon mentünk keresztül: véletlenül kinyitottam
```

1.1.4 kép: Saját programmal kinyerés helyes kimenete

Természetesen ez a módszer használatával is előfordultak hibák. Összevetve a saját program hibáinak típusait és mennyiségét az online konverterekével egyértelműen kilehetett jelteni, hogy a margón található szövegek levágásával jóval csökkent a rosszul tagolt mondatok és szavak száma.

A saját program írásának elkerülésére, az esetleges további hibák és a kinyerés precizitásának növelésére vetődött fel a fizetős konverterek használata. Egy ilyenre alkalmas program pl: az Adobe Reader [6]. Erre azonban nem került sor, ugyanis a Tesseract által generált szövegkimenet áttekintése után egyértelmű volt, hogy erre nem lesz szükség.

A saját program kimenetében rögtön kitűnt, hogy bizonyos szimbólumokat és írásjeleket nem ismer fel jól, mint pl. az idézőjeleket. Ezek helyére vagy egy sima vessző vagy dupla vessző került a kimenetben. Tekintve a 9.-es szöveggyűjtemény első oldalát ez előfordult 5 alkalommal. Ez az 5 alkalom közül 1 alkalom esetén az írásjel után következő szó is hibás lett. A szavak hibáit leszámítva ezek az írásjel típusú hibák nem befolyásolták a végső kimenetet, ugyanis az írásjelek a végső statisztikából kiszűrésre kerültek. További oldalakat tanulmányozva ez a hibatípus már okozott nagyobb gondokat, pl. a könyv 21. oldalán található rövid kérdez-felel párbeszéd esetén 2 szó elé is odakerült 1-1 's' betű, teljesen értelmezhetetlenné téve az adott szó. Bizonyos szavaknál, mint pl a 'nahát' szónál feltűnt, hogy ez a rendellenesség minden alkalommal előfordult, ahol ez a szó szerepelt. További oldalak tanulmányozása után egyértelműen kijelenthető volt, hogy a párbeszéddek amelyek minden sora idézőjellel kezdődtek legalább a nyitó idézőjel rossz volt, viszont szavak ritkán sérültek. Erre a hiba típusra nem történt automatizálható megoldás keresése, ugyanis a hiba változatossága és elenyésző befolyásoló faktora miatt úgy véltem, hogy nem érdemes vele foglalkozni.

1.2 MNSZ

A Magyar Nemzeti Szövegtárat (MNSZ) a Magyar Tudományos Akadémia Nyelvtudományi Intézetének Korpusznyelvészeti Osztályán kezdték el kialakítani 1998-ban Váradi Tamás vezetésével [7]. A cél egy 100 millió szavas szövegkorpusz kialakítása volt, amely a nagy adatmennyiség miatt reprezentatívan tartalmazta volna a magyar nyelv jellegzetes megnyilvánulásait. Az MNSZ manapság kb. 187.6 millió szövegszót tartalmaz, mindegyikhez tartozik szótő és szófaji elemzés is, ami kb. 97.5%-ban pontos. Az MNSZ által készített statisztikai adatok a saját eredményeink összehasonlításához fognak viszonyítási pontot adni.

	<i>szótő</i>	<i>szófaj</i>	<i>db</i>	<i>db / 1000 szó</i>
1.	a	<i>Det</i>	11128421	72,40
2.	az	<i>Det</i>	3716414	24,18
3.	és	<i>Con</i>	2544751	16,56
4.	hogy	<i>Con</i>	2166004	14,09
5.	A	<i>Det</i>	2103970	13,69
6.	az	<i>Pro</i>	1803814	11,74
7.	nem	<i>Adv</i>	1693748	11,02
8.	is	<i>Con</i>	1677108	10,91
9.	van	<i>V</i>	1418113	9,23
10.	ez	<i>Pro</i>	1204269	7,84
11.	egy	<i>Num</i>	899832	5,85
12.	Az	<i>Det</i>	730287	4,75
13.	meg	<i>Pre</i>	592986	3,86
14.	kell	<i>V</i>	499659	3,25
15.	csak	<i>Adv</i>	477956	3,11

1.2.1 kép: MNSZ által készített szógyakorisági táblázat részlete

2. Magyarlánc

A Magyarlánc a Szegedi Tudományegyetem által kifejlesztett magyar nyelv elemzésére szolgáló eszköz [8]. Java nyelven írt programról van szó, amely a beépített CLI-n keresztül történő használhatósága miatt könnyedén, jól konfigurálhatóan és platformfüggetlenül használható. A Magyarlánc fő célja a bemeneti magyar szövegnek a különböző szintű tokenekre bontása és azok elemzése [9]. A program több egymástól független modulra bontható, amelyeket bizonyos konfigurálhatósági paraméterekkel külön-külön vagy egyszerre is lehet alkalmazni a bemeneti szövegre:

- Sentence splitter
- Tokenizer
- Morphological analyzer
- POS tagger
- Dependency parser

2.1. Sentence Splitter & Tokenizer

Az elemzés legelső lépése, hogy a bemeneti szöveget a sentence splitter, mondatokra bontja. A magyarláncba épített sentence splitter és tokenizáló kifejlesztésére a MorphAdorner nevű nyelvi eszköznek a sentence splitterét és tokenizálóját vették és bővítették ki a magyar nyelv változatossága miatt. A sentence splitternek meg kellett tanítani, hogy pl. a ‘.’ karakter nem mindig jelzi a mondat végét. (erre példa mondjuk a ‘kft.’)

A mondatokra bontás után történik meg a tokenizálás. Természetesen ebben a fázisban is voltak olyan magyar nyelvi érdekességek, amikre az alap MorphAdorner tokenizáló nem volt felkészítve, mint pl: figyelnie kell a tokenizálónak, hogy a magyar nyelvben 2 vagy több egymás mellett álló betű lehet 1 hangot reprezentál, mint pl a ‘kocsi’ szó esetén, ahol mi megtanultuk, hogy a ‘cs’ betűket nem választhatjuk el egymástól, de ezt a tokenizálónak is tudnia kell kezelni.

2.2. Morphological analyzer

A tokenizálás lépés kimenetét ezután morfológiailag elemzi a program. Ehhez először a lemmatizáló modul szótári alakra alakítja a tokeneket, ebben a fázisban történik pl: a toldalékok eltávolítása. Emellett morfológiai kódok rendelődnek minden lemmához. Az analizáló modul eredete miatt KR kódokat készít, amit később konvertál át MSD kódokká [10].

2.3. POS Tagger

A szótári alakban levő tokenek elemzését ezután a Stanford POS-tagger egy módosított változata fogja végzi. A magyarláncba épített taggert a Szeged Corpus-on [11] tanították be, kezdetekben kisebb adathalmazzal, majd a végén pedig a teljes adathalmazzal hajtották végre. A kezdeti csökkentett adathalmaz fontos szinte minden gépi tanulási módszer használatakor, ugyanis az ilyen eszközök nincsenek arra felkészülve, hogy a legbonyolultabb bemeneteket is tudják kezelni már az elején.

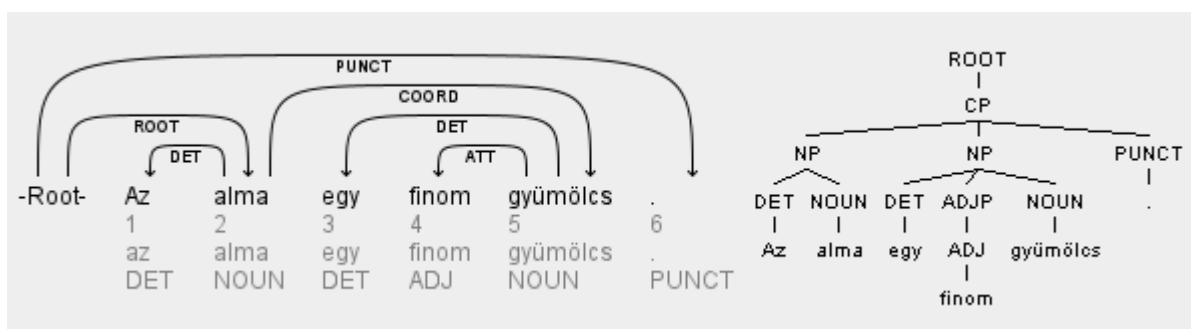
A POS tagging lépés fontos, ugyanis 1-1 adott token pozíciójának elemzése elárulja pl. azt, hogy a környezetében milyen szavak találhatóak, illetve azt is, hogy 1-1 adott szó a mondatban milyen szerepet tölt be. Ez mellett itt történik még a szavak szófajának megállapítása is.

2.4 Dependency parser

A szöveg értelmezésre általában 2 módszer az elterjedt: az egyik a konstituens nyelvtani elemzés, a másik pedig a függőségi nyelvtan elemzés. Mivel a magyar nyelvben a szavak pozíciója nincs megszabva, ezért a függőségi nyelvtani elemzés sokkal jobban és gyorsabban és pontosabban képes elemezni a magyar mondatokat, ugyanis az olyan egymással összefüggő szavakat is képesek egymáshoz kötni, amelyek nem egymás környezetében helyezkednek el. A magyarlánc ehhez a fázishoz a Bohnet elemzőt vették alapul, amelyet aztán a Szeged Dependency Treebank alapján tanítottak be. A Bohnet elemzőre azért esett választás, ugyanis több elemző tesztelése kimutatta, hogy ez volt a leghatékonyabb a magyar nyelv elemzésére. A Szeged Dependency Treebank egy kb. 82 ezer mondatból áll, amelyekben minden szó kézzel lett szófajilag elemezve.

2.5 Használat

A magyarláncot CLI-n keresztül többféle módban lehet használni, a különböző módok között a '-mode' parancssori paraméterrel lehet váltani. Egyes módok más elemzési folyamatokat alkalmaznak, ezért a kimenet és az elemzési sebesség a módok között nem megegyező. A magyarláncot lehet ablakos módban is futtatni a 'gui' '-mode' paraméter értékével. Ez a mód sokat segített megérteni és kideríteni, hogy mit is vár pontosan bemenetként a magyarlánc. A következő képen a 'Az alma egy finom gyümölcs' mondatának gui módban használt elemzésének kimenete láthatóak.



2.5.1 kép: A példamondat függőségi elemzése

2.6 Bemenet

Bemeneti fájlt ha parancssoros módban szeretnénk használni a ‘-input’ paraméteren keresztül kell megadni. Bemeneti formátum szempontjából a magyarlánc nem túl szigorú. Ha olyan szöveget/sort/mondatot talál amit nem tud elemezni, akkor egyszerűen kihagyja és halad tovább. Viszont előfordulhatnak olyan szövegtagolási érdekességek, amelyek képesek befolyásolni a kimenet pontosságát és helyességét.

Olyan esetre is akadt példa, hogy bizonyos bemenetekre az elemzés megszakadt és az alkalmazás kifagyott, ám ezt a kifagyást a nagyon hosszú elemzési idők és a kezdeti fázisban történő különböző módok váltogatása miatt nem sikerült reprodukálni.

2.7 Kimenet

```
Nyilván nyilván ADV _  
úgy úgy ADV PronType=Dem  
volt van VERB Definite=Ind|Mood=Ind|Number=Sing|Person=3|Tense=Past|VerbForm=Fin|Voice=Act  
, , PUNCT  
hogy hogy _SCONJ _  
szívszakadva szívszakadva ADV _  
vágytam vágyik VERB Definite=Ind|Mood=Ind|Number=Sing|Person=1|Tense=Past|VerbForm=Fin|Voice=Act  
a a DET Definite=Def|PronType=Art  
cirkuszba cirkusz NOUN Case=Ill|Number=Sing  
, , PUNCT _  
de de CONJ _
```

2.7.1 kép: Részlet a 9.-es szöveggyűjtemény elemzéséből

A kimeneti formátum a CSV formátum szerű, a különböző oszlopok tabulátor karakterekkel vannak elválasztva. Ez a kimeneti példa a -morphparse móddal készültek. Ez lényegében a legkevesebb elemzést végző mód, ugyanis ez a mód csak a szegmentálást és a POS Tagging lépéseket hajtja végre. Az első oszlopban az adott szó szövegbeli alakja szerepel, a másodikban az adott szó lemmája, a harmadikban a szófaj, majd végül az egyéb szófajhoz kapcsolódó jegyek.

3. Magyarlánc alternatíva

Alternatív szövegelemzők kipróbálására is került sor. A Stanford NLP csoport [12] által fejlesztett CoreNLP keretrendszer szintén Java nyelven íródott [13]. Több nyelvet is támogat, mindegyik nyelv külön-külön modulként hozzáadható a fő programhoz. A magyar nyelv támogatása a program által támogatott többi nyelvhez képest újnak számít, a 4.3.0-ás verziótól van rá támogatás.

3.1 Felépítés

Architektúráilag hasonló modulokból áll, mint a magyarlánc: ugyanúgy annotálják a szavakat pl. POS és az egyes szavak közötti egymás függési információkkal, viszont rendelkezésre áll olyan adat is amit a magyarláncnál nem található: képes pl. értelmezni, hogy egy-egy adott szó akár egy várost, céget vagy akár egy személy nevét reprezentálja (Named Entity Recognition).

3.2 Használat

Használhatóság szempontjából ugyanúgy lehet használni CLI-n keresztül, viszont dokumentált módon lehet programkódon keresztüli hívással is elemezteni szöveget, legyen szó akár Java nyelvről, vagy akár egy harmadik fél által szolgáltatott könyvtár segítségével szinte akármelyik manapság elterjedt programnyelven. Ezzel szemben a magyarlánc nem adott hivatalosan dokumentált módot programkódon keresztül történő használatra (természetesen ez nem kizáró ok), ám ezzel lehet vitatkozni, hogy a CLI miatt erre nincs is szükség.

3.3 Kimenet

Kimenet szempontjából a szófaji elemzések helyesek, viszont tokenizálási különbségek előfordulnak. Egy érdekesség ami rögtön kitűnt, hogy mondatok tagolása lépésben a CoreNLP nem ugyan úgy dolgozik, mint a magyarlánc: nincs megtanítva neki, hogy az olyan sorok, amelyek végén nincs írásjel külön mondatot jelenthetnek. Ez gondot okozhat pl. a címek elemzésénél, ahol a címek utolsó szavát akár teljesen egybe veheti a következő mondat első szavával. A következő képen látszik, hogy a 9.-es szöveggyűjteményben található ‘A cirkusz’ címsort nem tagolta külön mondatként a következőtől.

A	a	DET	
cirkusz	cirkusz	ADJ	
Nyilván	nyilván	PROPN	
úgy	úgy	ADV	
volt	volt	VERB	
,	,	PUNCT	
hogy	hogy	SCONJ	
szívszakadva	szívszakadva	ADV	

3.3.1 kép: Rossz mondattagolás példa

A lemma annotáció az említett verzióval tesztelve nem működött helyesen. A ‘szerződésről’ szóra a magyarlánc helyesen megállapítja, hogy a lemma a ‘szerződés’, míg a CoreNLP kimenetében a lemma ‘szerződésről’ marad, ami nem helyes. A kimenet áttekintése kiderült, hogy semelyik szó esetén nem történt meg helyesen a lemma megállapítása, mindenhol az eredeti szó maradt a lemma. (Ezt leszámítva a szófaj megállapítás viszont helyesen megtörténik)

A Named Entity Recognition egy olyan annotáció, amire a magyarlánc nem volt képes, viszont a CoreNLP igen. Tekintsük ezt a példamondatot: ‘Budapesten nagy a köd.’ A CoreNLP ebben a mondatban helyesen felismerte, hogy a ‘Budapesten’ egy helyszínt jelöl. Ez a felismerés kiterjed helyszínekre, nevekre stb.

3.4 Konklúzió

Tekintve az eredményeket a magyarlánc megbízhatóbbnak és pontosabbnak, ámbár lassabbnak bizonyult futási idő szempontjából a magyar szövegek elemzésére, mint a CoreNLP. Abban az esetben ha gyorsabban akarunk elemzéseket végezni és nincs szükségünk teljesen tökéletes elemzésekre (gondolva itt pl. a mondattagolási különbségekre), vagy pl. a tulajdonnevek elemzése érdekel minket, akkor a CoreNLP is egy teljesen megfelelő alternatíva.

4. Projekt

A projekt 2 alprojektből áll: szövegkinyerés és elemzés. Mind a 2 alprojekthez tartozik 1-1 java projekt és 1-1 F# nyelven írt szkript, amelyek a szövegkinyerés utáni tisztítást, illetve a nyelvi elemzők általi kimenet utóelemzését végzik. Ezen kívül tartozik a projekthez egy Python szkript ami a 2 alprojektek működéséhez szükséges fájlokat tölti le. (pl: magyarlánc, könyv pdf fájlok, Tesseract által használt betanítási fájlok)

A feldolgozás szinte minden fázisa egymástól független külön modulokba/projektekbe kerültek, elősegítve azt, ha valamelyik fázisban sikerülne valami javítást vagy fejlesztés véghezvinni akkor ne kelljen az egész kinyerés, tisztítás, elemzés folyamatot mindig újrafuttatni.

4.1 Szövegkinyerés

A szövegkinyerő program Java nyelven íródott a korábban már említett Tesseract nevű könyvtár használatával. A Tesseract bemenetként nem a komplett PDF fájlokat várja, hanem képeket. A program futása közben ezért oldalanként képek készülnek a PDF-ekből. Ez az extra lépés első ránézésre zavarónak tűnhet, viszont 2 fontos dolgot tesz lehetővé: az elemzendő kép manipulálását előfeldolgozás és elemzés előtt és az oldalankénti operáció miatt a teljesítmény növelését.

A generált képeken 1 módosítást végez a program mielőtt az elemzőnek tovább adná: a képek széleitől nézve levág egy kb. 1.5 cm-es margót, amely még az előfeldolgozás lépés előtt kényelmesen levágja a számunkra felesleges információkat, mint pl. az oldalszámokat és az ismétlődő főcímeket.

A teljesítmény növelése érdekében a program több oldalon dolgozik egyszerre, ugyanis az egyes oldalak elemzése teljesen független operációk. Ezzel a módszerrel 1 bemeneti pdf fájl esetén is gyorsabb elemzési időt érhetünk el, ellenben ha a másik úton indulunk el ahol egyszerre több pdf fájlra dolgozik akkor 1-1 adott pdf sebessége nem lenne gyorsabb. Első próbálkozásra ennek implementációja sikertelen volt, ugyanis a terminálban megjelenő hibaüzenetek arra mertek következtetni, hogy a Tesseract kinyerő kódja nincs felkészítve arra, hogy több szárról legyen használva. Ez könnyen megkerülhető volt azzal, hogy szálanként külön elemző objektum példány készül. A program jelenlegi implementációja egyszerre annyi oldalon dolgozik egyszerre, ahány logikai processzor található az adott rendszerben. Oldalanként nem történt időmérés, ugyanis 1-1 oldal elemzési ideje függ attól, hogy mennyi szöveg található azon az oldalon. A 8 PDF fájl elemzése így is közel 100 percet vesz igénybe 4 mag használatával, ahol a leglassabb a 12.-es szöveggyűjtemény volt 15 perccel, a leggyorsabb pedig a 9.-es tankönyv 7.5 perccel.

```
record ExtractResult(String content, int page) {}

private static ExtractResult extractTextFromPage(PDFRenderer imageRenderer, Tesseract tesseract, int page) {
    var image = imageRenderer.renderImageWithDPI(page, 300);
    var croppedImage = image.getSubimage(150, 150, image.getWidth() - 300, image.getHeight() - 300);

    return new ExtractResult(tesseract.doOCR(croppedImage), page);
}
```

4.1.1 kép: Részlet a szövegkinyerés Java kódból

Ez a függvény bemenetként vesz egy PDF fájlt és egy oldalszámot, csinál az adott oldalról egy 300 Dpi-s képet aminek minden oldalának széléből levág egy 150 pixelnyi margót. Erre a generált képre futtatjuk le a szövegkinyerést. A függvény tesseract paramétere az adott szálhoz tartozó tesseract objektum lesz, ugyanis ez a függvény fog egyszerre több szálon meghívódni. Az egyszerűség kedvéért a függvény hibakezelő részét kihagytam erről a képről. Bármilyen hiba esetén a program úgy veszi, hogy az adott oldal nem tartalmaz szöveget, viszont ilyen jellegű hiba nem fordult elő egyszer sem a könyvekből történő kinyerés alatt.

```
var extractedText = IntStream.range(0, inputPDFDocument.getNumberOfPages())
    .parallel()
    .mapToObj(i -> extractTextFromPage(imageRenderer, tesseract.get(), i))
    .sorted(Comparator.comparingInt(ExtractResult::page))
    .map(ExtractResult::content)
    .collect(Collectors.joining());
```

4.1.2 kép: Részlet a szövegkinyerés Java kódból

A fenti kódrészlet végzi az oldalankénti szövegek kinyerését, majd egyesítését. Mivel a kód egyszerre több szálon és meghatározott sorrend nélkül dolgozik az adott pdf oldalain ezért bevezetésre került egy ExtractResult nevű rekord ami a kinyert szöveg mellett még tárolja az adott oldal számát, lehetővé téve az oldalak újbóli sorbarendezeit. Ennek a részletnek a bemenete maga a pdf dokumentum, kimenete pedig a kinyert oldalak szövege egyesítve, ami aztán kerül kiírásra az adott könyv fájljába.

Az így kinyert szöveg már alkalmas lenne a magyarlánc általi elemzésre, viszont különböző próbálkozások kimutatták, hogy a magyarlánc kimenetét ronthatják olyan első ránézésre ártalmatlannak tűnő dolgok mint pl: ha egy mondat több sorra van tagolva, akkor bizonyos szavak nem helyesen elemződnek, vagy akár bizonyos írásjelek teljesen egyesülnek elemzéskor a körülöttük előforduló szavakkal. Ennek a hibának a mitigálására lett bevezetve egy előfeldolgozó szkript.

```
Máskor idegen,
nagy város közepén egyszerre ott álltam, de ugyanaz a cirkusz volt, ugyanaz a bejárat, kétfelé
nyíló vesztibülő.
```

4.1.3 kép: Részlet a szövegkinyerés kimenetéből

Máskor	máskor	ADV	PronType=Ind
idegen,	idegen,	PROPN	Case=Nom Number=Sing
nagy	nagy	ADJ	Case=Nom Degree=Pos Number=Sing
város	város	NOUN	Case=Nom Number=Sing

4.1.4 kép: Részlet a példamondat javítás előtti elemzéséből

A fenti képeken a 9.-es szöveggyűjteményből vett részlet és annak elemzése látható. Az elemzésre ránézve is látszik, hogy nem minden helyes: hibásan 2 mondatra szedte a magyarlánc, majd hozzávette az 'idegen' szóhoz a vesszőt, ami révén a szófaj névmássá alakult. A mondat 1 sorra rendezése után már helyes az elemzés.

Máskor	máskor	ADV	PronType=Ind
idegen	idegen	ADJ	Case=Nom Degree=Pos Number=Sing
,	,	PUNCT	
nagy	nagy	ADJ	Case=Nom Degree=Pos Number=Sing
város	város	NOUN	Case=Nom Number=Sing

4.1.5 kép: Részlet a példamondat javítás utáni elemzéséből

4.2 Előfeldolgozás

Minden tisztítási folyamat előtt felvetődik az a kérdés, hogy érdemes lenne-e automatizálható, programkódbeli megoldást keresni rá vagy kézzel is megoldható-e az adott tisztítás. Sok olyan szöveg elem van, ami nem ad semmit az elemzéshez:

Ilyenek pl a margókon található oldalszámok/folyamatosan ismétlődő címek. Ezek kiszűrésére kezdetben regex alapú szűrés volt tervben, de az utólagos szűrés helyett úgy döntöttem, hogy érdemesebb más szögből megközelíteni a problémát, azaz el se tárolni azt a szöveget már a szövegkinyeréskor ami nem kell.

A kinyerési folyamat után történt az egyedüli felhasználói beavatkozás: a könyvek elején/végén található felesleges szövegek (bevezető részek, tartalom és fogalomtárak) eltávolítását kézzel végeztem az automatizált tisztító szkriptek futtatása előtt.

A több sorra tagolt mondat hibának kezelése szinte triviális, ugyanis egyszerű szövegcsereeléssel az ilyen több sorra tagolt mondatok könnyedén egyesíthetőek. Erre írtam egy kis egyszerű előfeldolgozó szkriptet, amely az olyan sorokat amelyek lezáratlanok újraegyesíti.

```

let paraphraseFileContent file =
    file |> File.ReadAllText
        |> fun k -> k.Replace(Environment.NewLine, "\n")
        |> fun k -> k.Split "\n\n"
        |> Seq.map(fun k -> k.Replace("-\n", "").Replace("\n", " "))
        |> fun k -> String.Join("\n", k)

```

4.2.1 kép: Részlet az előfeldolgozó F# kódból

A fenti kódrészlet feladata a bemeneti fájl beolvasás, szétagolása, formázása és egyesítése. Mivel a szövegkinyerés kimenetében a nagyobb szövegblokkok 2 újsor karakterrel vannak tagolva, a szkript ezeket veszi paragrafusoknak. Az ezeken belül található szövegen belül fogja lecserélni az olyan sorvégződések, amelyek arra utalhatnak, hogy az adott mondat még folytatódna a következő soron, ezzel egyesítve azokat. A függvény végezetül fogja a formázott szöveget és paragrafusonként egyesíti őket, közéjük 1-1 újsor karaktert téve. Természetesen a formázás végén 1-1 soron belül szerepelhet több mondat, de ez a magyarlánc számára nem probléma, hiszen gond csak akkor van, ha 1 mondat több sorra tagolódik.

4.3 Nyelvi elemzés

Ebben a lépésben került használatra a magyarlánc. Hivatalos dokumentáció szerint parancssorból kéne használni, viszont ennek 1 hátránya van, amit az ablakos mód használata közben fedeztem fel: CLI-n keresztül minden egyes meghívott fájl előtt a magyarlánc inicializálás újra megtörténik, amihez nem kevés idő és memória szükséges. Mivel a magyarlánc forrásai online elérhetőek [14], ezért ennek megkerülésére hozzáadtam a magyarláncot mint függőség a Java projekthez és a programon belül hívtam meg kézzel a magyarlánc inicializáló és elemző függvényeit, ezzel elérve azt, hogy több bemeneti fájlra is le tudjam futtatni az elemzést úgy, hogy csak egyetlen egyszer kelljen inicializálni, ezzel jelentős mennyiségű időt spórolva a különböző tesztelések között. A szöveget a mi esetünkben a magyarlánc 'morphparse' módjával szeretnénk elemezni, amihez a programunkban az ehhez a módhoz megfelelő Magyarlanc.morphParse elemző függvényt kell használnunk.

```

private static void analyzeWithMagyarlanc(Path inputFile, Path outputDirPath) {
    try(var output = Files.newBufferedWriter(Path.of(outputDirPath.toString() + '/' + inputFile.getFileName()))) {
        var input = SafeReader.read(inputFile.toString(), StandardCharsets.UTF_8.name());
        var sentences = Magyarlanc.morphParse(input);

        for(var sentence : sentences) {
            for(var tags : sentence) {
                output.write(tags[0] + '\t' + tags[1] + '\t' + tags[2] + '\n');
            }

            output.write('\n');
        }
    }
}

```

4.3.1 kép: Részlet a magyarláncal elemző Java kódból

Ez a kód fog lefutni minden bemeneti fájlra. Bemenetként a bemeneti fájlt és a kimeneti mappát várja. Az egyszerűség kedvéért ebből a kódból is kihagytam a hibakezelést és az output writer paraméterezését. A try with resource blokk végén a kimeneti fájl bezáródik, amikor is íródik majd bele az elemzés kimenete. A kimenet formázását a CLI-n keresztül használat kimenetéhez alakítottam, annyi változtatással, hogy amire nem volt szükség azt nem írja ki a program a kimenetbe.

A sebesség növelése érdekében természetesen felmerült itt is a többszálúság, azaz egyszerre több fájlra meghívni a magyarláncot, viszont a terminálban megjelenő hibaüzenetek arra mertek következtetni, hogy a magyarlanc nincs arra felkészítve, hogy több szálról legyen egyszerre meghívva.

```

Caused by: java.lang.IndexOutOfBoundsException: start 155, end 154, length 169
    at java.base/java.lang.AbstractStringBuilder.checkRange(AbstractStringBuilder.java:1802)
    at java.base/java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:680)
    at java.base/java.lang.StringBuilder.append(StringBuilder.java:214)
    at java.base/java.util.regex.Matcher.appendReplacement(Matcher.java:999)
    at java.base/java.util.regex.Matcher.replaceAll(Matcher.java:1181)
    at splitter.utils.PatternReplacer.replace(PatternReplacer.java:84)
    at splitter.ling.tokenizer.AbstractPreTokenizer.pretokenize(AbstractPreTokenizer.java:141)
    at splitter.ling.tokenizer.DefaultWordTokenizer.extractWords(DefaultWordTokenizer.java:45)
    at splitter.ling.sentencesplitter.AbstractSentenceSplitter.extractSentences(AbstractSentenceSplitter.java:501)
    at splitter.MySplitter.split(MySplitter.java:251)
    at splitter.MySplitter.splitToArray(MySplitter.java:235)
    at hu.u_szeged.magyarlanc.Magyarlanc.morphParse(Magyarlanc.java:126)

```

4.3.1 kép: Részlet a hibakimenetből

Ez a hiba nem volt determinisztikus: futásonként változott a hiba helye, viszont a kimenetben fellelhető közös pont alapján sikerült leszűkíteni a hiba egy potenciális okozóját: a magyarlanc kódjában található 'PatternReplacer' nevű osztályában a 'sourcePatternMatcher' nevű mezőjének a 'reset' nevű függvényét használná a program esetünkben egyszerre több szálról szinkronizálás nélkül a 'replace' nevű metódusában:

```
protected Matcher sourcePatternMatcher;

public String replace(String s) {
    return sourcePatternMatcher.reset(s).replaceAll(replacementPattern);
}
```

4.3.2 kép: Részlet a magyarulanc 'PatternReplacer' osztályából

Ebben a metódusban a sourcePatternMatcher.reset több szálon történő hívása esetén a 'Matcher' objektum több szálon használdik egyszerre, amire a Matcher objektum nincs felkészítve [15]. Ebben az esetben is történt próbálkozás a hiba megkerülésére, viszont itt nem lehetett azzal a módszerrel megoldani, mint a szövegkinyerési folyamatot, ugyanis a hiba feltárása után egyértelművé vált, hogy a kód felépítése ezt nem teszi lehetővé. Mivel java-ban van lehetőség futás közbeni kódmanipulációra, az említett metódus lecserélése és javítása után az említett hiba megszűnt, viszont a hiba típusából egyértelműen gondolni lehetett arra, hogy máshol is fognak hasonló típusú hibák jelentkezni, ami be is következett. Ennek a javításnak a kódja a magyarulancot futtató projektben megtalálható kikommentelve.

Természetesen ez a többszálúságú gondolkodás a CLI módszert használva működhetett volna egyéb beavatkozás nélkül is (ugyanis ekkor a magyarulanc több folyamatként futna egymástól teljesen függetlenül), viszont a korlátozott rendelkezésre álló feldolgozási erő miatt ez se nem lett volna praktikus, se nem kivitelezhető.

Ebben a projektben van még egy CoreNLP-t használó elemző program is, amelynek kimenete ugyanolyan formátumra lett hozva, mint a magyarulancé, ezzel lehetővé téve a későbbi elemző kódok egységes használatát.

```
private static void analyzeWithCoreNLP(Path inputFile, Path outputDirPath, StanfordCoreNLP pipeline) {
    try(var output = Files.newBufferedWriter(Path.of(outputDirPath.toString() + '/' + inputFile.getFileName())));
    var document = new CoreDocument(Files.readString(inputFile));
    pipeline.annotate(document);

    for(var sentence : document.sentences()) {
        var posTags = sentence.posTags();

        for(var token : sentence.tokens()) {
            output.write(token.word() + "\t" + token.lemma() + "\t" + posTags.get(token.index() - 1) + "\n");
        }

        output.append('\n');
    }
}
```

4.3.2 kép: Részlet a CoreNLP-vel elemző Java kódból

A fenti képen látható függvény nagyon hasonlít a magyarulccal elemző program kódjára. Itt a kimenet formázását pont úgy végeztem el, ahogy azt a magyarulccénál is tettem, ezzel lehetővé téve az ezután következő kódok kompatibilitását. A további eredmények előállításához a már korábban említett okok miatt a CoreNLP kimenete nem került felhasználásra, csak a magyarulcc által elemzéseinek a kimenetei.

4.4 Utóelemző

A végső eredményeket a nyelvi elemzők futtatása után egy F# szkript-el alakítottam ki. A kimenetet különböző diagram és szövegfelhő [16] képfájlok, illetve az ezek generálásához használt adatokból készített json fájlok képezik. Ezek a kimeneti fájlok a magyarulcc kimenetét nézve könyvenkénti lebontásban és egyesített formában kerültek kimentésre.

A program a különböző adatformátumok feldolgozására 2 külön függvénybe van szervezve: egyik a magyarulcc kimenetét, a másik pedig a teljes MNSZ gyakorisági listát alakítja át egy olyan adatszerkezetté, ami mind a 2 formátumból előállítható: minden szóhoz hozzárendelt gyakoriság és szófaj. Ez az MNSZ adatainál triviális, minden szó már el van látva gyakorisági számokkal és szófajokkal, míg a magyarulcc által generált fájlokban kódból kellett összeszámoltatni a szavakat. Mind a 2 feldolgozást megnehezítette az a tény, hogy bizonyos szavaknak több szófajuk is volt a kimenetben. Ez a későbbi kimenetek előállításánál volt különösen zavaró, ugyanis a program ezen pontjától minden más adat csak ebből az egy szógyakorisági adatokból állt elő.

```
let parseMagyarlancWordFrequencies file =
    let calculatePOSCounts(args: seq<string[]>) = args |> Seq.countBy(fun k -> k.[2]) |> dict

    file |> File.ReadLines
        |> Seq.filter(fun k -> k <> String.Empty)
        |> Seq.map(fun k -> k.Split '\t')
        |> Seq.filter(fun k -> not(Array.contains k.[2] magyarlancAnalysisIgnoredPOSeS))
        |> Seq.groupBy(fun k -> k.[0])
        |> Seq.map(fun (word, args) -> { | word = word; posCounts = calculatePOSCounts args | })
        |> Seq.toArray
```

4.4.1 kép: Magyarlanc kimeneti fájljának szógyakorisági elemzését előállító függvény

A fenti képen látszik, hogy semmi különösét nem kellett csinálni a kimenettel. Egyedül egy szófaji szűrő került be, ami a nem túl érdekes és hibás szófajokat szűri ki a teljes kimenetből. A program következő fázisában a szógyakorisági adatokból állítódnak elő a további adatok.


```

let createAnalysisStat(rawWordFrequencies: { | word: string; posFrequencies: IDictionary<string, int> | }[]) =
    let createPOSStat pos =
        let wordsToPOSFrequency = rawWordFrequencies |> Seq.filter(fun k -> k.posFrequencies.ContainsKey pos)
        |> Seq.map(fun k -> (k.word, k.posFrequencies.[pos]))
        |> dict

        let mostFrequentWords = wordsToPOSFrequency |> Seq.map(|KeyValue|)
        |> Seq.sortByDescending(fun (_, freq) -> freq)

        let longestWords = wordsToPOSFrequency.Keys |> Seq.map(fun k -> (k, k.Length))
        |> Seq.sortByDescending(fun (_, length) -> length)
        {
            frequency = wordsToPOSFrequency |> Seq.sumBy(fun (KeyValue(_, freq)) -> freq)
            averageWordLength = wordsToPOSFrequency |> Seq.averageBy(fun (KeyValue(word, _)) -> float word.Length)
            mostFrequentWords = mostFrequentWords |> dict
            longestWords = longestWords |> dict
        }

    let posStats = rawWordFrequencies |> Seq.collect(fun k -> k.posFrequencies.Keys)
    |> Seq.distinct
    |> Seq.map(fun pos -> (pos, createPOSStat pos))
    |> Seq.sortByDescending(fun (_, stats) -> stats.frequency)

    let wordFrequencies = rawWordFrequencies |> Seq.map(fun k -> (k.word, k.posFrequencies)) |> dict

    {
        wordCounts = createWordCountStat wordFrequencies
        wordFrequencies = wordFrequencies
        wordLengths = rawWordFrequencies |> Seq.map(fun k -> (k.word, k.word.Length)) |> dict
        posStats = posStats |> dict
    }

```

4.4.2 kép: A szavak gyakorisági adatait előállító függvény

Ez a függvény bemenetként a korábban említett feldolgozó függvények kimenetét várja paraméterül, és 4 adatot ad vissza: szavak számát, minden szóhoz hozzárendelt gyakoriságot, minden szóhoz hozzárendelt hosszúságot, illetve szófajonkénti lebontásban azok darabszámát, az átlag szó hosszát, a leggyakoribb és leghosszabb szavak listáját. Az első 2 előállítása nagyon egyszerű, szimplán minden szóhoz hozzárendeli a darabszámot/szóhosszt. A szófajonkénti adatok előállítása már érdekesebb: ide került bevezetésre egy segédfüggvény, ami 1 db szófajt vár paraméterül és a hozzá tartozó már említett adatokat számolja ki. Ehhez először összeszedi az olyan szavakat, amiknek a szófajlistájában szerepel az adott szófaj és az ahhoz a szófajhoz tartozó darabszámot rendeli a szóhoz. Ebből a többi adat előállítása szinte triviális: a szófajgyakoriság az összes szó darabszámának összege, az átlag szóhossz az összes szó hosszának átlaga, leggyakoribb szavak listája a szavak listájának gyakoriság szerinti sorbarendezeése, illetve a leghosszabb szavak listája pedig szóhossz szerinti sorbarendezees. egymástól jól elkülöníthető részre oszlik. Ez a segédfüggvény lesz meghívva minden szófajra, majd a végkimenetben az adott szófaj darabszáma alapján kerül sorbarendezeesre.


```

let emptyStats = { wordFrequencies = dict([]); wordLengths = dict([]); posStats = dict([]) }

let perBookStats = Directory.GetFiles magyarlancOutputDir
    |> PSeq.map(fun k -> (Path.GetFileNameWithoutExtension k, k |> parseMagyarlancWordFrequencies |> createAnalysisStat))
    |> PSeq.toArray

let perBookTypeStats = perBookStats |> Seq.groupBy(fun (fileName, _) -> parseBookType fileName)
    |> dict

let szgyStats = perBookTypeStats["szgy"] |> Seq.map(fun (_, stats) -> stats)
    |> Seq.fold mergeAnalysisStats emptyStats

let tkStats = perBookTypeStats["tk"] |> Seq.map(fun (_, stats) -> stats)
    |> Seq.fold mergeAnalysisStats emptyStats

let mergedStats = mergeAnalysisStats szgyStats tkStats

```

4.4.3 kép: Elemző függvény használata, további adatformátumok előállítása

A fent látható képen az előbb említett függvény kerül használatra, megtörténik a könyvenkénti feldolgozás. A teljes feldolgozás minden könyvre egyszer történik meg, a további más formában lebontott elemzések elkészítéséhez a könyvenként előállított adatok kerülnek egyesítésre.

Az MNSZ adatainak feldolgozása a programon belül opcionális rész, ugyanis az adatforrás beszerzéséhez felhasználói input kell. Mivel az adatformátum eltér a magyarlancétól, ezért az újrahasználhatóság szempontjából feldolgozásakor más oszlopszámokat kellett használni, illetve a szófajokat normalizálni kellett. Emellett be kellett vezetni egy regex alapú szószűrőt is, ugyanis az 500MB-os adathalmaz tömeges része teljesen értelmetlen szavakkal és szimbólumokkal van tele. A következő képen az MNSZ adataira lefutó json fájl és diagram/szófelhő előállító kód látható. Ez a kód nagyon hasonlít a magyarlanc kódjához, viszont külön lett tőle szedve, ugyanis a kimenet nem teljesen egyezik meg.

```

let mnszAnalysisStat = mnszDataFilePath |> parseMNSZWordFrequencies |> createAnalysisStat

File.WriteAllText($"{statsOutputDir}/mnsz.json", JsonSerializer.Serialize(mnszAnalysisStat |> truncateAnalysisStat, statJsonSettings))
wordCloudPOses |> Seq.iter(writeWordCloud mnszAnalysisStat.wordFrequencies $"{wordCloudOutputDir}/mnsz")

mnszAnalysisStat.posStats |> Seq.filter(fun (KeyValue(pos, _)) -> not(Array.contains pos posChartIgnoredMNSZPOses))
    |> Seq.map(fun (KeyValue(pos, stat)) -> (pos, stat.frequency))
    |> Seq.sortByDescending(fun (_, freq) -> freq)
    |> dict
    |> writeChart $"{chartOutputDir}/mnsz_pos_distribution.png" CHART_PIE "Szófaj"

```

4.4.4 kép: MNSZ adatainak teljes feldolgozó kódja

Míg a szófelhők előállítását ugyan ebben a szkriptben, addig különböző diagramok generálását egy python szkript meghívásával végzem. Ennek szimpla oka az, hogy python-ban már volt általam ismert könyvtár ami ezt tudta (matplotlib). A diagram előállításához szükséges adat json formátumra hozva kerül átadásra a generáló szkriptnek.

```
chart_args = json.loads(argv[1])
chart_type = chart_args['chartType']
data = chart_args['data']
ax = pyplot.subplots()[1]

if chart_type == 'pie':
    edges = ax.pie(data.values(), autopct = '%1.0f%%')[0]
    ax.legend(edges, data.keys(), title = chart_args['legendTitle'], loc = 'center left', bbox_to_anchor = (1, 0, 0.5, 1))

pyplot.savefig(chart_args['filePath'])
```

4.4.5 kép: Kördiagram előállításáért felelős kódrészlet

A generált diagramok és szófelhők konkrét számszerű adatait a már korábban említett json fájlokba menti ki a program. Ezekbe a fájlokba nem a teljes adathalmaz kerül kiíratásra, ugyanis a teljes adathalmaz felhasználása teljesen felesleges lett volna, a grafikonok és szófelhők előállításához pedig megoldhatatlan. A leggyakoribb szavak halmazból az első 100 leggyakoribb szó, szófajonkénti lebontásban az első 5, szófelhők generálásához pedig a leggyakoribb 500 szó kerül felhasználásra/kiíratásra.

```
"wordFrequencies":
  "a": {
    "DET": 48115,
    "PRON": 1,
    "PROPN": 8
  },
  "az": {
    "PRON": 1134,
    "DET": 15345,
    "PROPN": 1
  },
  "és": {
    "CONJ": 13513,
    "INTJ": 5
  },
  "A": {
    "DET": 8921,
    "PROPN": 25,
    "NOUN": 3
  },
  "is": {
    "CONJ": 5456,
    "ADV": 1076,
    "SCONJ": 23,
    "PROPN": 1
  },

"posStats": {
  "NOUN": {
    "frequency": 194822,
    "averageWordLength": 8.852876179091869,
    "mostFrequentWords": {
      "ember": 907,
      "vers": 726,
      "szöveg": 552,
      "költő": 479,
      "O": 426
    },
    "longestWords": {
      "budapest-páris-berlin-kamcsatka-szentpétervár": 45,
      "ínyvitorla-szívlapát-lélegzethinta-mérleg": 41,
      "Nemvoltrendszeresesztétikaiképzettsége": 38,
      "ezernyolcszázhetven-egynéhányban": 32,
      "hagyományőrzéshagyománytagadás": 30
    }
  },
}
```

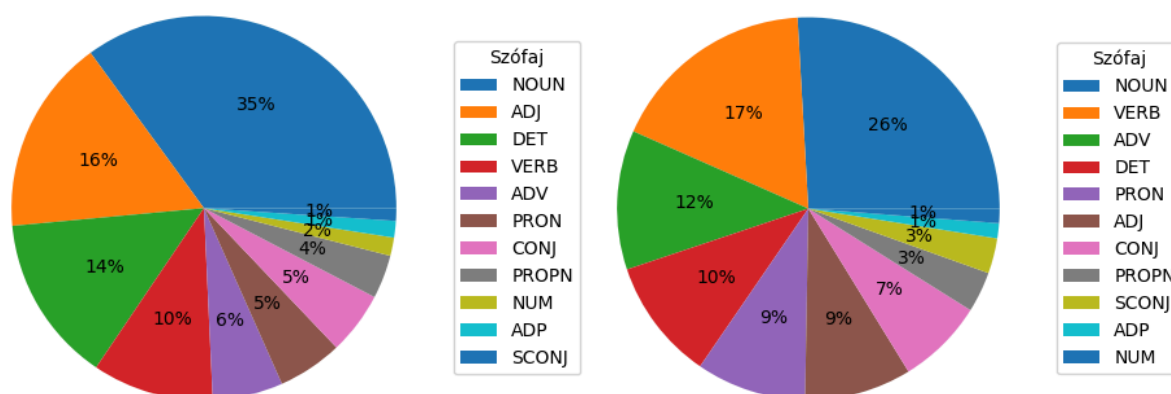
4.4.6 és 4.4.7 képek: Részletek a merged.json fájlból

5. Eredmények

Eredményeimet különböző kritériumok alapján csoportosítottam: szófaji eloszlások évfolyamonként, tankönyvek és szöveggyűjtemények összesítve, illetve az összes könyv összesítve. Fontos megjegyezni, hogy az MNSZ adataira generált diagramok/grafikonok/egyéb adatok pontossága kétségbevonható, ugyanis az adathalmaz nagyon sok olyan szót/szófajt tartalmaz, amiknek szinte jelentésük sincs. A diagramokon a láthatóság növelése érdekében az ilyen érthetetlen szófajoknak egy bizonyos része kiszűrésre került.

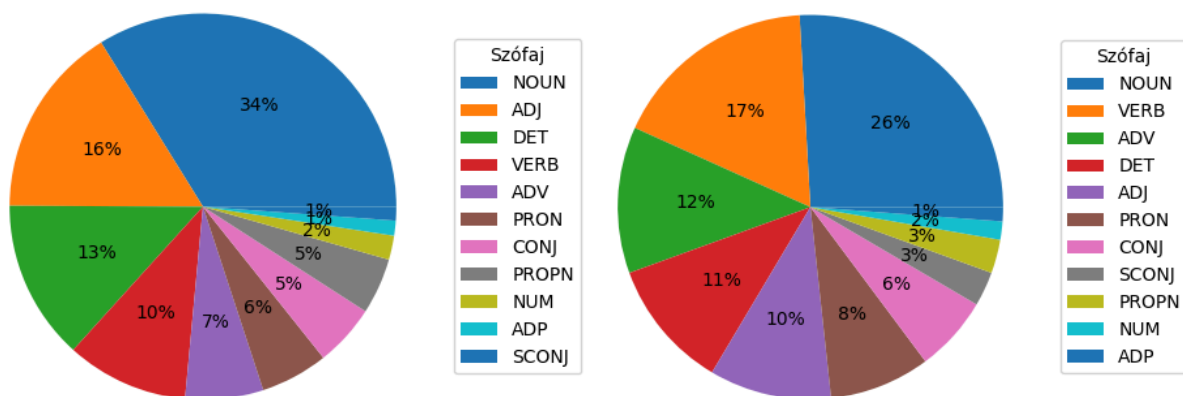
5.1: Szófaji eloszlások

A tankönyvek és szöveggyűjtemények szófaji eloszlásaik között fellelhetőek szabályosságok: a szófajok eloszlásánál megfigyelhető, hogy a szöveggyűjtemények arányaiban kevesebb főnevet és több igét tartalmaznak. Vegyük például csak a 9.es könyveknek az adatait: a következő 2 grafikonon bal oldalt a tankönyvé, jobb oldalt pedig a szöveggyűjtemény szófaji eloszlásai láthatóak.



5.1.1 és 5.1.2 kép: 9.-es könyvek szófaji eloszlásai

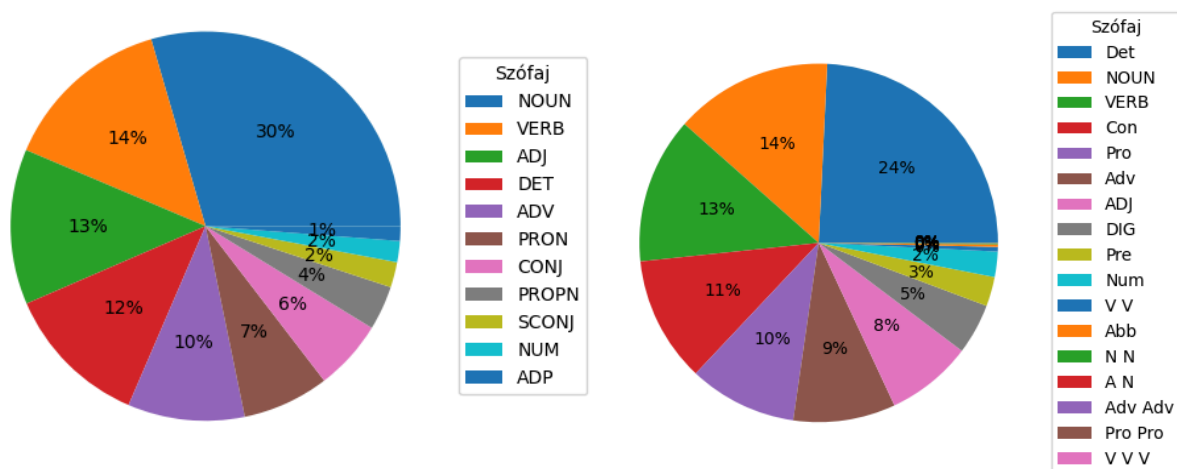
A tankönyv esetében a főnevek a szavak 35%-át tette ki (15536 a 44758-ból), míg a szöveggyűjtemény esetében csak 26%-ot (21056 a 82044-ből). Az igéknél is megfigyelhető ilyen mértékű különbség: a tankönyvnél csak 10% volt (4522 a 44758-ból), míg a szöveggyűjtemény esetében 17% (14212 a 82044-ből). A 12.-es könyvek esetén ezek a grafikonok nagyon hasonlóak: A tankönyvnél elenyésző 1%-os eltérések vannak, míg a szöveggyűjtemény esetén ebben az esetben a főnevek és igék eloszlása teljesen ugyanaz volt. Ezt ha a tankönyvek és szöveggyűjtemények összesített adataira vetítjük ki akkor is hasonló jelenséget figyelhetünk meg.



5.1.3 és 5.1.4 kép: Könyvek összesített szófaji eloszlásai

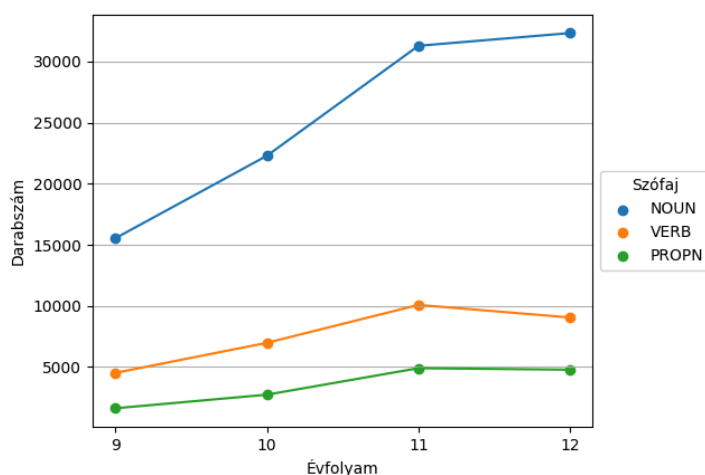
Ezeknek a szabályosságoknak az lehet az oka, hogy a szöveggyűjtemények úgy általában több szót, és több olyan mondatot (úgy általában több mondatot is, az írásjelek százalékos eloszlásaik is erre utalhatnak) tartalmaznak amiben történések leírása van, nem pedig pl. fogalmak tárgyalása.

Ha a könyvek összesített szófaji eloszlásait hasonlítjuk össze az MNSZ szavainak szófaji eloszlásával, akkor megfigyelhető, hogy az MNSZ esetében az igék és a főnevek arányai között nincs olyan mértékű különbség, mint a könyvek esetén. Amit személy szerint érdekesnek tartottam, hogy az MNSZ esetében a teljes szavak 24%-át (21450971 a 96116576-ból) tették ki a határozószavak, ami ebben az esetben a szavak többségét jelentette, míg a könyvek esetén ezen szavak aránya mindig jóval kisebb volt. (Itt az összesített esetén 12% (79424 a 666282-ből), de a különböző lebontásokban is ettől 1-2%-al tért el csak az arány)



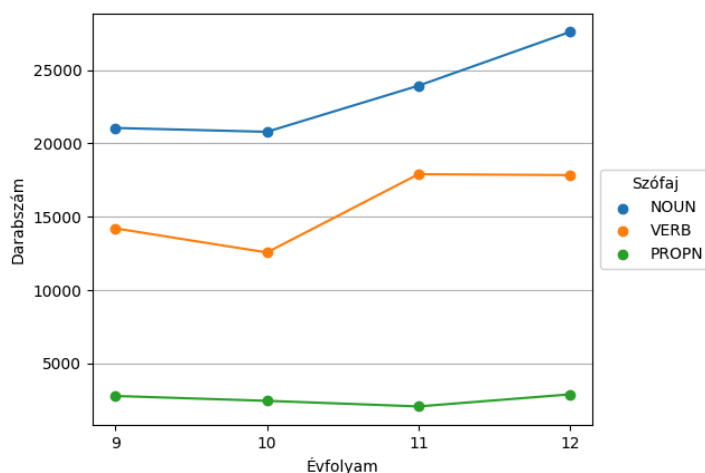
5.1.5 és 5.1.6 kép: Könyvek összesített szófaji eloszlásai

A különböző évfolyamok esetén is fellelhetőek drasztikus változások a tankönyvek esetén: a főnevek száma haladva az évfolyamokkal folyamatosan nőtt (kezdetben csak 15536, a végén már 32325), viszont pl. az igék száma nem mutatott ilyen mértékű drasztikus növekedést, ahol pl. a 11 és a 12-es könyvek között még csökkenés is történt. Ez mellett még a tulajdonnevek száma is jelentősen megnőtt, a 9.-es tankönyvben csak 1624, míg a 12.-es esetén közel 3x ennyi (4780).



5.0.7 kép: Tankönyvek esetén nézett igék és főnevek eloszlása évfolyamonként

A szöveggyűjtemények ebből a szempontból jóval kiegyensúlyozottabbak voltak, ott nem történtek ilyen mértékű növekedések/csökkenések. Az igék és főnevek száma is növekedett évfolyamonként folyamatosan, viszont a tulajdonnevek száma szinte változatlan maradt. Ez mellett szinte semmilyen szabályosság/folyamatosság nem volt megfigyelhető.



5.0.6 kép: Szöveggyűjtemények esetén nézett igék és főnevek eloszlása évfolyamonként

5.2: Szavak hossza és számuk

A különböző adatbázisok esetében összesített adatleltárban nem volt érdemes nézni a szavak hosszúságának átlagát/leghosszabb szavakat, ugyanis mind az MNSZ adatbázisa, mind a könyvekre generált adatbázis tartalmazott olyan érthetetlen és nagyon hosszú szavakat, amik ezeket az adatokat nagyon torzították volna. 1-1 szófajon belül viszont érdemes volt elemezni: a legrövidebb szavakat egyértelműen a határozószavak tették ki, az MNSZ esetében 1.75-ös, a könyvek esetében pedig 2.33-as átlaghosszúsággal. A könyvek egyesített elemzésében a főnevek átlaghossza 8.85-volt, a mellékneveké 8.36, az igééké 8.27. Az MNSZ átlagaival összevetve (főnevek: 10.47, igék: 10.26, melléknevek: 10.17) látszik, hogy az MNSZ-ben valamivel hosszabb szavak szerepelnek. Ehhez sajnos nem lehet levonni érdemlegesebb következtetést a már korábban említett nagyon hosszú értelmetlen szavak torzításai miatt, véleményem szerint ezek a számok hasonlóak, annyi látszik általánosságban, hogy a főnevek átlaghossza valamivel nagyobb, mint az igék átlaghossza.

Ami érdekesség volt, hogy a könyvek külön-külön nézése esetén a tankönyvek és szöveggyűjtemények között már egyértelműen látszottak különbségek. A tankönyvek esetén a főnevek átlaghossza 9.5 volt, míg a szöveggyűjteményeknél: 8.2. Ezt ha külön-külön könyvenként is nézzük, akkor egyértelműen látszik, hogy a tankönyvekben egy picivel átlagosan hosszabb főnevek vannak. A könyvenkénti lebontások esetén nem voltak kilógó értékek, az összes könyv esetén 0.1 és 0.3-nyi különbségek voltak. Az igéknél ilyen szabályosság nem szerepelt, mint a tankönyvek és a szöveggyűjtemények esetén a már korábban említett átlaghoz képest elhanyagolható mennyiségű eltérések voltak. A melléknevek esetén is hasonlóak voltak az eredmények.

Szavak mennyiségét vizsgálva 2 lebontás tűnt érdekesnek: az összesített szavak száma és az egyedi szavak száma. Mind a kettő esetében megfigyelhető volt az a jelenség, hogy az osztály növekedésével egyre több szóból álltak a könyvek, illetve egyre több egyedi szavat is tartalmaztak. Az utóbbit a nagyobb elvárt szókinccsel lehet magyarázni.

	Szöveggyűjtemény	Tankönyv	Összesített
9. osztály	82044	44758	126802
10. osztály	76853	67687	144540
11. osztály	98582	95167	193749
12. osztály	106364	94827	201191
Összesített	363843	302439	666282

5.2.1 kép: Összesített szavak száma

Érdekes volt pl. a 9. osztály esetében az, hogy a szöveggyűjtemény összesített szavainak száma majdnem 2x annyi, mint az ugyanazon évfolyam tankönyvé. Ez a jelenség ezután évfolyamonként a 12. osztályig folyamatosan eltűnik, ahol már szinte majdnem megegyeznek a szószámok. Ha az összesített szöveggyűjtemény és tankönyv szószámokat figyeljük látszik, hogy a szöveggyűjtemények kb. 60 ezerrel több szóból állnak.

	Szöveggyűjtemény	Tankönyv	Összesített
9. osztály	25730	14742	
10. osztály	27789	20671	
11. osztály	30313	28473	
12. osztály	34931	28065	
Összesített	85913	64582	131301

5.2.2 kép: Egyedi szavak száma

Az egyedi szavak számát nézve is a szöveggyűjtemények tartalmaznak több szót. Itt is megfigyelhető az a jelenség, hogy a szöveggyűjtemények és tankönyvek szókincsei a 12. osztályra körülbelül kiegyenlítődnek. Itt annyi különbség lelhető fel, hogy az összesített szavak számából a szöveggyűjtemények arányaiban nagyobb részt tesznek ki, mintha az összesített szavak számát néznénk.

Megjegyzés: Az egyedi szavak osztályonkénti lebontásban nem érhetőek el, ugyanis ilyen lebontásban nem készültek elemzések.

5.3: Szófelhők, szógyakoriságok



5.3.1 kép: Könyvek összességére generált szófelhő

	Szó	Gyakoriság		Szó	Gyakoriság
1.	a	48124	11.	Az	3063
2.	az	16480	12.	volt	2542
3.	és	13518	13.	el	2236
4.	A	8949	14.	mint	2141
5.	is	6556	15.	de	2102
6.	hogy	6407	16.	csak	2069
7.	nem	6021	17.	már	1996
8.	s	4514	18.	vagy	1867
9.	meg	4321	19.	ki	1855
10.	egy	4007	20.	még	1630

5.3.2 kép: Összesített könyvek első 20 leggyakoribb szavai

A fenti szófelhő képén látszik, hogy a szűrés nélkül készült felhőknek túl sok érdekességük nincs, ugyanis a szógyakorisági listák leggyakoribb szavait nézve mind az MNSZ, külön-külön az összes könyvre és azok egyesített elemzéseikre is elmondhatóak, hogy kb. az első 15-20 szó szinte ugyan azokon a helyeken szerepelnek, szinte ugyanolyan eloszlási arányokkal. (Mivel az MNSZ adatbázisában megkülönböztetésre került pl. a kisbetűs ‘a’ és a mondat elején szereplő nagybetűs ‘A’ ezért ez a jelenség az általam kimentett adatokban is megfigyelhető)

Ez a probléma láttán lettek generálva a főnevekre, igékre és mellénevekre. Ha a fontosabb szófajokra generált képeket nézzük, itt már egyértelműen kivehetők az irodalom témához tartozó szavak. Vegyük például a főnevekre generált szófelhőt:



33

Az összesített elemzésben a leggyakoribb főnév az **ember** szó volt, ami 907 szer szerepelt a könyvekben, amiből 548 a szöveggyűjteményekben volt. Erről a szóról érdemes megjegyezni, hogy az MNSZ adatbázisában is ez volt a leggyakoribb főnév szó, ott 117163-szor szerepelt. A második helyen a **vers** szó szerepelt 726 db előfordulással, amiből 701 a tankönyvekben volt. Ez természetesen várható volt, ugyanis a tankönyvekben főleg versek elemzéseiről van szó. Ha a további leggyakoribb főneveket összevetjük a tankönyvek és szöveggyűjtemények között, akkor egyértelműen látszik, hogy a témák különbözősége miatt a tankönyvekben főleg versek és egyéb irodalmi művek elemzéseivel kapcsolatos szavak vannak többségben, míg a szöveggyűjteményekben pedig a magyar epikának a tipikus szavai, mint pl. **nap**, **föld**, **asszony**. Az MNSZ 5 leggyakoribb főnevei közül az **ember** szó kivételével egy sem került bele a könyvek 5 leggyakoribb főnevei közé.

Az igékre generált szófelhőket nem volt érdemes összehasonlítani. Az MNSZ adataival összevetve nem jelentek meg a témához kapcsolódó szavak, mind a 2 lista legelején a **volt**, **van**, **kell** és **volna** szavak szerepeltek. Ez az általánosság nem csak az összesített elemzésekre mondhatók el, hanem a könyvenkénti lebontású elemzésekre is.

Mellékneveknél a könyvek és az MNSZ felhői között fellelhetőek voltak hasonlóságok és különbségek is egyaránt:



5.4.4 és 5.4.5 kép: Bal oldalt a könyvekre, jobb oldalt az MNSZ melléknevek felhői

Az MNSZ adataiban a leggyakoribb melléknév a **magyar** szó volt, ahol kb. 30 ezerszer szerepelt. A bal oldali felhőn is látszik, hogy ez a szó a könyvekben is benne van az első 5-ben, ott a kb. 85 ezer melléknévből közel 800 csak a **magyar** szó volt. Hasonló eredményeket mutatott fel a **nagy** szó is, ami a mind a könyvekben, mind az MNSZ adataiban a 2. helyen szerepelt. A könyvekben a leggyakoribb melléknév a **című** szó volt, ez a szó az MNSZ adatai között szinte fellelhetetlen.

6. Összegzés

Összességében sok érdekességet tudhattam meg a szövegkinyerés folyamatáról és a szövegek szótani elemzéséről. A folyamat különböző fázisain egyértelműen volna potenciál az eredmények pontosságának növelésére (pl. a szövegkinyerő rész hibázási lehetőségeinek minimalizálásával), illetve újabb adathalmazok létrehozására.

Elemzéseim végén arra jutottam, hogy egyértelműen érdemes foglalkozni a különböző tankönyvek szótani elemzéseivel. Érdekes lenne megfigyelni esetleg más nyelvű tankönyvek, vagy akár más tantárgyú könyvek elemzéseit is.

Irodalomjegyzék

1. Hazánkban használt szókinccsmérő eljárások I. (Juhász Valéria – Radics Márta) https://anyanyelv-pedagogia.hu/img/keptar/2019_1/Anyv_XII_2019_1_3.pdf
 2. Korszerű, használható tankönyveket az iskolákba (Oktatási Minisztérium 2006. február) <http://www.nefmi.gov.hu/letolt/kozokt/ktvrendelet.ppt>
 3. Tankönyvek és segédletek online katalógusa | 2021 – 2022 (Oktatási Hivatal) <https://www.tankonyvkatalogus.hu>
 4. PDFForge szövegkinyerő: <https://tools.pdfforge.org/extract-text>
 5. Tesseract: <https://github.com/tesseract-ocr/tesseract>
 6. Adobe Reader OCR: <https://www.adobe.com/acrobat/how-to/ocr-software-convert-pdf-to-text.html>
 7. Magyar Nemzeti Szövegtár (MTA Nyelvtudományi Intézet) <http://mnsz.nytud.hu>
 8. magyarlanc: A Toolkit for Morphological and Dependency Parsing of Hungarian (Department of Informatics, University of Szeged) <https://aclanthology.org/R13-1099.pdf>
 9. magyarlanc: a toolkit for linguistic processing of Hungarian (MTA-SZTE Research Group on Artificial Intelligence) <https://rgai.inf.u-szeged.hu/magyarlanc>
 10. Konverterek magyar morfológiai címkészletek között (Vadász Noémi, Simon Eszter MTA Nyelvtudományi intézet) <http://real.mtak.hu/101696/1/99-111.pdf>
 11. Szeged Corpus 2.5: Morphological Modifications in a Manually POS-tagged Hungarian Corpus http://www.lrec-conf.org/proceedings/lrec2014/pdf/262_Paper.pdf
 12. The Stanford NLP Group <https://nlp.stanford.edu>
 13. CoreNLP (Stanford NLP Group) <https://stanfordnlp.github.io/CoreNLP>
 14. Github/zsibritajanos/magyarlanc <https://github.com/zsibritajanos/magyarlanc>
 15. Thread-safety with regular expressions in Java (Neil Coffey) https://www.javamex.com/tutorials/regular_expressions/thread_safety.shtml
 16. Tag cloud (Wikipedia) https://en.wikipedia.org/wiki/Tag_cloud
-
9. szöveggyűjtemény: https://www.tankonyvkatalogus.hu/storage/pdf/OH-MIR09SZ_teljes.pdf
 9. tankönyv: https://www.tankonyvkatalogus.hu/storage/pdf/OH-MIR09TA_teljes.pdf
 10. szöveggyűjtemény: https://www.tankonyvkatalogus.hu/storage/pdf/FI-501021002_1_teljes.pdf
 10. tankönyv: https://www.tankonyvkatalogus.hu/storage/pdf/FI-501021001_1_teljes.pdf
 11. szöveggyűjtemény: https://www.tankonyvkatalogus.hu/storage/pdf/FI-501021102_1_teljes.pdf
 11. tankönyv: https://www.tankonyvkatalogus.hu/storage/pdf/FI-501021101_1_teljes.pdf
 12. szöveggyűjtemény: https://www.tankonyvkatalogus.hu/storage/pdf/FI-501021202_1_teljes.pdf
 12. tankönyv: https://www.tankonyvkatalogus.hu/storage/pdf/FI-501021201_1_teljes.pdf

Nyilatkozat

Alulírott Gyúróczki Gergő Viktor programtervező informatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Számítógépes Algoritmusok és Mesterséges Intelligencia Tanszékén készítettem, programtervező informatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Diplomamunka Repozitóriumban tárolja.

Dátum 2025. márc. 24

Köszönetnyilvánítás

Ezen módon is szeretném megköszönni Dr. Csirik János témavezetőm segítségét, aki a rendszeres konzultációk alatt biztosította iránymutatásával szakdolgozatom zökkenőmentes elkészültét, hasznos ötletekkel látott el, és legfontosabban is: kitartásra ösztönzött.

Elektronikus melléklet

Szakdolgozatom mellékletét a MagyarSzovegekThesis.zip fájl képezi, amelyben a következő struktúrában helyeztem el a projekteket és a kimeneteket:

- **setup.py:** A projekthez szükséges fájlokat tölti le (könyvek .pdf fájljai, magyarlanc.jar) és a projektek helyes működéséhez szükséges beállításokat konfigurálja.
- **magyarlanc_src.zip:** A 10-es irodalomjegyzék pontban említett Github repozitórium .zip exportja, ez a forráskódot tartalmazó .zip fájl került felhasználásra a fejlesztés alatt.
- **Analyzers mappa:** A különböző elemző programok (magyarlanc, CoreNLP) futtatására szolgáló projektet és a hozzátartozó .json elemzési eredmények és szófelhő generáló szkriptet tartalmazza.
- **TextExtractor mappa:** A .pdf fájlokból történő szövegkinyeréssel foglalkozó projektnek a kódját tartalmazza. Ez mellett tartalmazza még a kinyerés után futtatott tisztító szkriptet is.
- **outputs mappa:**
 - **text mappa:** 2 mappát tartalmaz: a raw nevezetűben a .pdf fájlokból kinyert nyers szöveg van, a paraphrased nevezetűben pedig a nyers szövegre lefuttatott tisztító szkriptnek a kimenete.
 - **analyze mappa:** .txt formában tartalmazza a CoreNLP és a magyarlanc kimeneteit a hozzájuk tartalmazó mappában. Ez mellett ebbe a mappába kell elhelyezni az MNSZ adatbázisának fájlját is hnc-1.3-wordfreq.txt néven.
 - **stats mappa:** A magyarlanc kimenetére és az MNSZ adatbázisára lefuttatott elemző szkript statisztikai kimenetét tartalmazza .json formátumban könyvenként és egyesítve.
 - **word_cloud mappa:** A magyarlanc kimenetére és az MNSZ adataira lefuttatott elemző szkript szófelhő kimeneteit tartalmazza .jpg formátumban. Itt bizonyos fájloknak a végén egy ‘_SZÓFAJ’ végződés van, amelyek csak az adott szófajra lettek generálva. A nem ilyen végződésű képek a teljes szólistára lettek generálva.
 - **chart mappa:** A különböző szófaji eloszlások és szófajok számának könyvenkénti változásainak diagramjait tartalmazza.

A projektek és szkriptek beüzemeléséhez szükséges dolgok:

- Python telepítés (tesztelve Python 3.9.6-al)
- Java telepítés (tesztelve Java 17-el és 18-al)
- Maven telepítés (tesztelve 3.8.1-el)
- Dotnet telepítés (tesztelve Dotnet 5.0.404-el)

A projektek és szkriptek futtatása:

- Először futtatni kell a setup.py fájlt (terminálban 'python setup.py')
- A szövegkinyerő és az elemző projektek Maven projektek, érdemes őket beimportálni egy IDE-be, mint pl. Eclipse-be.
- A szövegkinyerő projekten belül a 'textextractor.Main' nevű fájlt kell futtatni a szövegkinyerés indításához.
- A szövegkinyerés után kell lefuttatni a Paraphrise.fsx fájlt. (terminálban 'dotnet fsi Paraphrise.fsx')
- Az elemző projekten belül 2 futtatható fájl van: A magyarlánccal elemző a 'analyzers.MagyarlancRunner' fájl, a CoreNLP-vel elemző pedig a 'analyzers.CoreNLPRunner' fájl. Az 'analyzers.CustomPatternReplacer' fájl használatához az 'analyzers.MagyarlancRunner' fájlban belül kell kieszni a 20-26. sorig tartó kommentezést. (Ez a korábban említett parallelizáláshoz tartozó javítás, ennek a tesztelésére a 35. soron kezdődő Java Stream-et paralell módra kell kapcsolni)
- Ahhoz, hogy a következő lépésben az MNSZ adataira is lefussanak az elemzések, kézzel le kell tölteni a teljes gyakorisági listát az <http://mnsz.nytud.hu/> oldalról, kibontani a zip fájlt, majd a benne található .txt fájlt a outputs/analyze mappába tenni.
- Az elemzések lefutása után kell lefuttatni a Analyze.fsx fájlt. (terminálban 'dotnet fsi Analyze.fsx')
- Ezek után a kimenet az előbb említett mappákba kerül. A text/raw mappa tartalma nem fog megegyezni újrafuttatás után a kézzel történt könyvek elején és végén található szövegek eltávolítása miatt. Ez mellett a word_cloud/magyarlanc tartalma nem fog megegyezni, ugyanis a szavak pozícionálása és színezése véletlenszerűen történik a képek generálásakor.