Tudásalapú adatelemző rendszer természetes nyelvű interfésszel

Budapesti Műszaki és Gazdaságtudományi Egyetem

Méréstechnika és Információs Rendszerek Tanszék

Szerző: Rabinovits Jakov

Konzulens: Dr. Mészáros Tamás Csaba

2017. október 27.

Tartalomjegyzék

Bevezető

Manapság egyre szélesebb körben használnak számítógépes adatelemzési eszközöket. Munkám során két aspektusra helyezek hangsúlyt, melyek napjainkban egyre relevánsabbak. Egyrészt a szöveg elemző- illetve kereső rendszerek hatékonysága sok területen nem teljesíti az elvárásokat, másrészt a felhasználók jelentős része nem rendelkezik mélyebb informatikai ismerettel, ami sokszor szükséges az említett eszközök használatához. Ilyen alkalmazási területek például a természet tudományok és a bölcsészet, ahol a rendelkezésre álló adatok minősége sokszor nem megfelelő a jelenleg elterjedt eszközök számára.

Dolgozatomban a problémák megoldására a szakterületi tudás jobb kiaknázásával teszek javaslatot, ennek keretében egy tudásalapú szövegelemző-rendszert fejlesztettem, amely rendelkezik egy természetes nyelvű interfésszel is.

A rendszer lehetővé teszi a betöltött szövegek tudásalapú feldolgozását, amelynek a megvalósításához szükséges szemantikus információt RDF formátumban tárolja. A dolgozatom során megmutatom, hogy a klasszikus kulcsszó alapú kereséssel szemben ez a módszer jobb pontosságú találatokat eredményez, valamint a szövegelemzési és más hagyományos stilometriai műveletek során is jobb eredményeket ér el.

A rendszer másik előnye a természetes nyelvi interfész, amely lefordítja a lekérdezéseket a rendszer által feldolgozható belső parancsokra. Ez megkönnyíti azok munkáját, akik nem rendelkeznek programozási ismeretekkel. Munkám során példaként a szövegelemző-rendszerhez az irodalmárok által használt stilisztikai eszközöket nyújtó felületet valósítottam meg.

Elméleti háttér

Keresés

A jelenlegi kulcsszó alapú keresők megszokott és kényelmes módot biztosítanak információ szerzéshez, viszont nem tudják értelmezni ezen a kulcsszavak jelentését [1]. Ezzel szemben az ontológia alapú keresők jobb eredményeket érnek el, de azok használata nehezebb. Míg az előbbi kategóriánál elég kulcsszavakat beírni, az utóbbi esetben ismerni kell az adatok szintaxisát. A kérések megfogalmazása is bonyolultabbá válik, hiszen olyan formális nyelveket kell használni, mint a SPARQL.

A szemantikus web koncepciója az, hogy tegyük lehetővé az egész web szemantikus leírását. A mostani kulcsszó alapú webes keresők javarészt statisztikai alapúak. Nagy fejlődésen mentek keresztül, így az információbeszerzés hatékonyságában felülmúlják a szemantikus keresőket. Viszont, ha a hagyományos keresőket szeretnénk használni domén specifikus környezetekben, akkor az eredmények általában nem megfelelő minőségűek. Ennek az az oka, hogy azok korlátozottabb információ mennyiséggel, ugyanakkor sok rendhagyó kifejezéssel rendelkeznek.

Például egészségügyi területen, ami elég nagy ontológiai bázissal rendelkezik, szemantikus keresés és fejlett NLP (natural language processing) eszközök lehetőséget kínálnak természetes nyelven megfogalmazott, tudással bővített keresések végrehajtásához. Az egyik kutatás [2] eredményeképpen készítettek egy interfészt, ami a természetes nyelvi kéréseket automatikusan lefordította szemantikus keresésekre, amik már tartalmazzák a megfelelő ontológiát.

A kutatás során a felhasználók tényeket és kérdéseket fogalmaztak meg, amire a kereső egészségügyi eszközöket javasolt. Az ontológiai kereső jobb eredményt ért el, mint a hagyományos, kulcsszó alapú keresők.

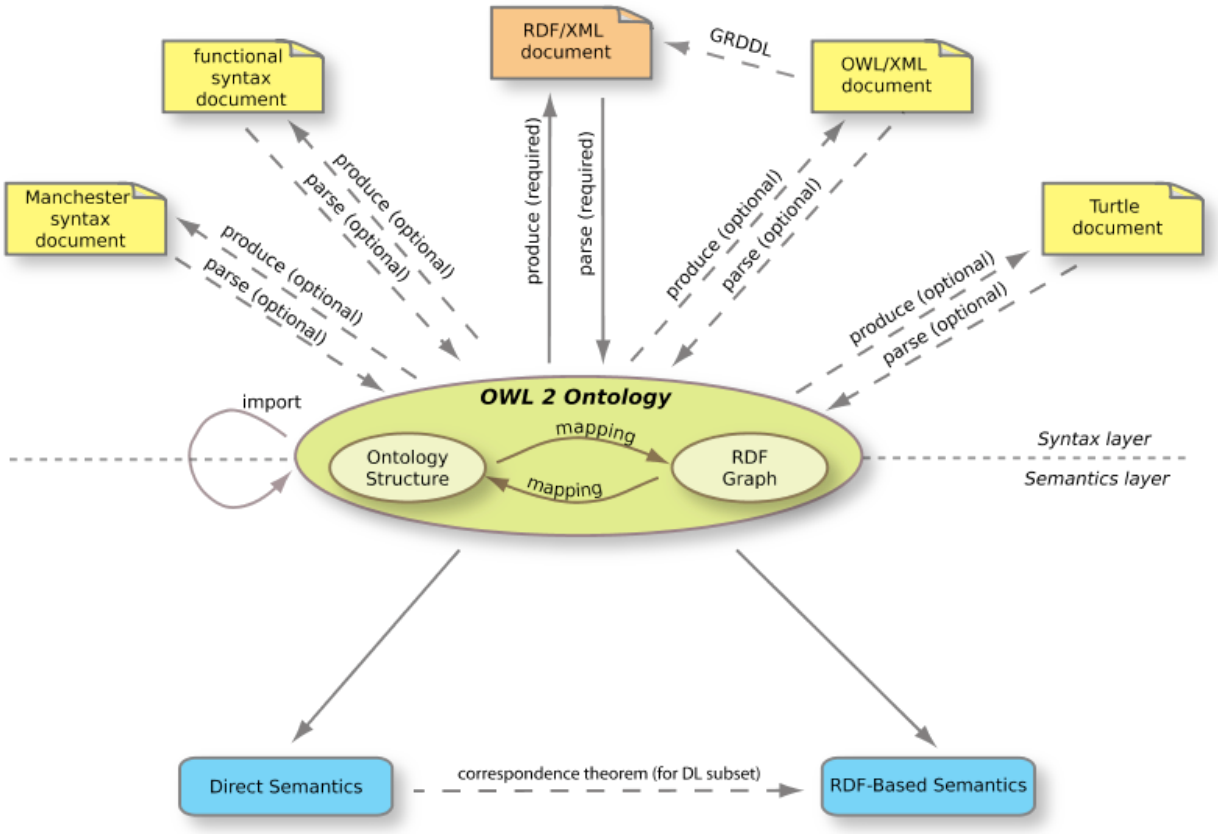
Szemantika definiálása: OWL/RDF

Az OWL nyelvet [3] arra tervezték, hogy az információt ne csak emberek számára lehessen közvetíteni, hanem lehetővé tegye az alkalmazások számára a szövegek szemantikus

elemzését. Ezzel jobban támogatja a webes tartalom gépi interpretációját, mint az egyszerűbb RDF illetve XML sémák.

Az OWL nyelvet a W3C csoport fejleszti annak a céljából, hogy a jövőben a webes tartalomnak legyen explicit szemantikus információja. Így könnyebben lehetne azt feldolgozni, és automatikusan integrálni információt más webes erőforrásokból. Ez a szemantikus web koncepciója.

A szemantikus web XML formátumot használja fel, mivel azzal tetszőleges adatokat rugalmas módon le lehet írni. Az XML önmagában csak a szintaxist határozza meg a strukturált dokumentumok számára, nem szab meg szemantikus korlátokat.



1. ábra. OWL felépítése [3]

Az RDF formátum egy adatmodell a webes objektumok (erőforrások) számára [4]. Meghatározza az objektumok közötti kapcsolatokat és azok szemantikáját. Ezt XML formátum segítségével írja le. Az RDF séma definiálja az RDF erőforrások osztályait és tulajdonságait.

Az OWL ezen felül még több szintaxist biztosít a tulajdonságok és osztályok, illetve az osztályok közötti kapcsolatok leírására. Így például, ki lehet fejezni az osztályok kardinalitását, osztály hierarchiákat és más jellemzőket.

A W3C ajánlása alapján az OWL a tudás reprezentálásához az RDF gráfokat használja, amelyek tripletekből állnak. Ezek a tripletek három komponensből tevődnek össze:

- Subject, ami egy azonosító vagy egy üres csomópont
- Predicate, ami egy azonosító
- Object, ami egy azonosító, egy literális vagy egy üres csomópont

Látszik, hogy ez a gráf struktúra egy bonyolultabb formátum, mint a nyers szöveg. Ezért az extra komplexitásért cserébe szemantikus jelentést tulajdonítunk egyes erőforrásoknak.

Jelenlegi megoldások

Az alábbiakban bemutatok két eszközt, ami megpróbál általánosabb módon nyújtani szemantikus keresést és ontológiák használatát. Mindkettő Apache Solr keresőmotorra épül, annak indexelését használja az RDF/OWL formátumú ontológiák használatához.

Open Semantic Search

Open Semantic Search [5] egy kutatási eszköz szövegelemzéshez, szemantikus kereséshez és szövegbányászathoz. Interfészt biztosít ontológiák és szótárak használatához. Ezen kívül, rendelkezik adatvizualizációs és interaktív keresési eszközökkel. Az utóbbinál ontológiák alkalmazásával jeleníti meg a szövegek metaadatait, helységek és tulajdonnevek felismerését. A felsoroltakon kívül még számos műveletet biztosít.

Az Open Semantic Search telepítése és beüzemeltetése egyszerű. Elég egy előre konfigurált virtuális gépet vagy egy debian csomagot letölteni. Viszont, két problémát is felfedeztem az eszközzel kapcsolatban. Egyrészt, nem rendelkezik megfelelő interfésszel külső alkalmazások felé. Néhány REST API-s végpontot definiál, de azok meglehetősen korlátozottak, nagyon kevés műveletet ajánlanak fel. A keresési eredmények lekérését, ontológiák és szótárak

feltöltését nem teszi lehetővé külső alkalmazások számára. Csak saját grafikus felületet biztosít ezen műveletek elvégzéséhez.

Továbbá, az ontológiák és szótárak törlése, módosítása sokszor súlyos hibákat idéz elő, amitől leáll az egész rendszer. Több környezetben is próbáltam, de a használata nem volt stabil.

Azonban a grafikus felülete felhasználó barát, tehát ha nem szeretnénk szerver alkalmazást írni, ami kommunikálna ezzel a szoftverrel, helyette csak egyszerű felhasználóként tekintünk rá, akkor egy opció lehetne.

Apache Stanbol

Apache Stanbol egy Solr alapú nyílt forráskodú projekt [6]. Több, független komponensből áll, amik mind Apache projektek.

A komponensei az alábbi funkcionalitásokat biztosítanak:

- Szemantikus információt adhatunk hozzá a nyers szövegekhez
- A szolgáltatások további információt nyújtanak a tartalomról a megadott szemantikus információ alapján
- A szolgáltatások tudásalapú adatmodelleket definiálnak és manipulálnak
- Ezeket az adatokat perzisztens módon tárolják, így lehetővé teszik a szemantikus keresést

Stanbol segítségével tetszőleges dokumentumokat lehet indexelni. Az eredményeket pedig RDF és JSON formában adja vissza.

Stanbolt különálló alkalmazásként és web szerverként is lehet használni. Emellett bő REST interfészt biztosít, ami alkalmassá tenné egy saját szerver alkalmazással való kommunikációra.

Viszont, a Stanbol projekttel két fő problémám volt. Egyrészt, körülményes a használata, ami az informatikában kevésbé jártas felhasználók számára gondot jelent. Másrészt, a beüzemeltetése során számos hibával kellett küzdeni, és végül az összes komponenst nem sikerült

beüzemeltetnem, mivel az új, stabil verzió nem rendelkezik azokkal. A régebbi verziót pedig nem sikerült feltelepíteni.

Tapasztalatok

Ezek az eszközök alapvetően intelligensek, számos funkciót nyújtanak, de sok nehézséggel és hibával rendelkeznek. Az említett okokból kiindulva saját rendszerbe próbáltam integrálni a szemantikus eszközöket. Egyelőre egy egyszerű szótár alkalmazását tettem lehetővé, ami már nagy mértékben tudja javítani a szövegek elemzését, illetve a keresési műveleteket. Jövőben a rendszert ki lehet egészíteni ontológiák beépítésével, amivel még hatékonyabbá és intelligensebbé lehetne tenni a szemantikus műveleket.

Szövegbányászati eszközök

A szoftverem másik alappillére az adatbányászati eszközök használata.

Manapság egyre nagyobb mértékben jelenik meg az igény adatbányászati eszközök alkalmazása iránt. Az alkalmazásuk nagy jelentőséggel bír tudományos és vállalati világban egyaránt. A dolgozatom során elsődlegesen szövegbányászatra koncentrálok, ami az adatbányászat egyik ága.

Szövegbányászat során tudást vagy jelentős mintákat próbálunk nyerni a strukturálatlan szövegekből. Ehhez számos területet kell alkalmazni, mint szöveg elemzés, kategóriázálás, vizualizáció, gépi tanulás és egyebek. Ezek alapvetően bonyolult műveletek, de mára már megjelent számos eszköz, amelyek a segítségünkre lehetnek.

RapidMiner, R

Jelenleg az adatbányászat legnépszerűbb eszközeit a RapidMiner és R programozási nyelv jelentik [7]. Az alábbiakban áttekintem a két eszközt.

R egy programozási nyelv, a legnépszerűbb grafikus fejlesztői környezete az RStudio. RapidMiner pedig egy adattudományos szoftver platform, amit Javaban készítettek. Mivel R egy programozási nyelv, így rugalmasabban használható, mint a RapidMiner. R több formátumú

fájlal tud dolgozni és több vizualizációs eszközt nyújt. R nyelvhez könnyen készíthetünk új csomagokat, amivel kiterjeszthetjük az eszköztárat.

Viszont, az RStudioval ellentétben, a RapidMiner telepítése és használata egyszerűbb. RapidMiner használatához nincs szükség programozási nyelvhez, tehát kisebb képzettségi szintet igényel. Ez egy fontos szempont azok számára, akik nem rendelkeznek mélyebb informatikai tudással.

Mivel R bővebb funkcionalitással rendelkezik, így azt választottam a saját rendszer megalkotásánál. Mint később kiderül, egy természetes nyelvi interfész segítségével egyszerűbbé tehetjük ennek az eszköznek a használatát.

Saját rendszer

Az előbb említett technológiák és kész szoftverek néhány jellemzőben hiányosak. Egyrészt, azok beüzemeltetése általában nem egy könnyű feladat, és gyakran előfordulnak olyan hibák, amik miatt leáll a szolgáltatás vagy el sem indul.

Másrészt, az laikus felhasználók számára ezek a szoftverek használata körülményes.

Ezekből a megfontolásokból én is elkezdtem egy szemantikus rendszer fejlesztését. Ennek a rendszernek az egyik előnye, hogy rendelkezik egy természetes nyelvi interfésszel. A másik előnye az, hogy integrálja a keresést és a szövegelemzési műveleteket.

Áttekintés

Ebben a részben röviden ismertetem a megoldásomat, majd kitérek a felhasznált technológiákra, végül pedig részletesen bemutatom a rendszer működését.

Az alapvető ötlet az, hogy webszervert készítünk, ami biztosít néhány végpontot REST API-val, amin keresztül le lehet kérni és végrehajtani az összes műveletet. A felhasználó kategorizálni tudja a dokumentumokat (ezek lesznek a Lucene indexek, amikre később kitérek), fel tudja tölteni a dokumentumokat fájlokból vagy manuálisan beszúrt szövegekből, amiket a szerver oldalon eltároljuk.

Az eltárolt szövegekben hagyományos módon lehet keresni kulcsszavak alapján. A szerver visszaadja egyrészt azokat a dokumentumokat, amikben megtalálta a megadott kulcsszavakat, másrészt kiemeli azokat a részeket, ahol szerepelt a kulcsszó.

Ahhoz, hogy jobb eredményeket érjünk el a kereséssel, lehetőség van a megadott indexekhez feltölteni szinonima szótárakat. Így, a keresésnél nem csak a megadott terminust fogja keresni a rendszer, hanem annak szinonimáit is. Ezzel már egy egyszerű szemantikus jelentést tulajdoníthatunk a dokumentumoknak.

Továbbá, lehetőséget biztosítunk ontológiák használatához. Ehhez egy RDF fájlt kell feltölteni, ami tartalmazza a tudást. Lucene indexek rugalmasságának köszöntően, azokat is meg lehet

indexelni, azokon is végezhetünk keresést. Így, össze lehet kapcsolni a nyers szöveget az RDF formátumú gráfokkal.

Ezen felül, a rendszer biztosít néhány alapvető szövegelemző és stilisztikai műveletet. Ilyen műveletek a stop szavak törlése, whitespace karakterek eltüntetése, és ezekhez hasonló, szokásos műveletek. Emellett, lehet generálni egy szófelhőt. Ezt fogom alkalmazni az eredmények szemléletes bemutatásához.

A műveleteket a megadott indexeken lehet elvégezni, és minden művelet elvégzése után a felhasználó megkapja az indexekben lévő dokumentumokat jelenlegi állapotát. Természetesen ezeket a eredményeket is fel lehet tölteni a szerverre.

A rendszer különlegessége az, hogy az említett műveletek elvégzéséhez biztosít egy természetes nyelvi interfészt. Miután megkaptuk a kérést, azt feldolgozzuk és utána egyszerűen átirányítjuk a megfelelő végpontra. Ilyen módon továbbra is van lehetőség azokat közvetlen módon alkalmazni, nem korlátozzuk a klienst olyan módon, hogy a természetes nyelvi interfészt kelljen alkalmaznia. Ennek fényében készíthetünk egy olyan webes klienst, ahol szokványos módon, grafikus módon el tudjuk érni ugyanazokat a funkciókat. Jelenleg ilyen webes felület még nem áll rendelkezésre, csak REST API végpontokat valósítottam meg.

Használt technológiák

Alább bemutatom a rendszerem által használt technológiákat, meglévő szoftvereket, amik segítettek a legfontosabb kihívások legyőzésében. Ebben a részben csak a használt technológiákat ismertetem röviden, ezek működésére és egymással való kapcsolatára a következő részben térek ki.

Elasticsearch (Lucene)

A keresőmotornak az Elasticsearch projektet választottam [8]. Ez egy nyílt forráskódú Javaban írt projekt. Solrhoz hasonlóan ez is egy Lucene alapú keresőmotor. Viszont, az utóbbival szemben, Elasticsearch sokkal rugalmasabb és bővebb REST API-val rendelkezik, illetve van hozzá egy fejlett Python interfész. A dokumentumokat JSON formátumban tárolja, sémák

nélkül, ami megkönnyíti a kisebb szövegek kezelését. Ezen az okok miatt döntöttem végül úgy, hogy az Elasticsearchet használok.

Elasticsearch további előnyei, hogy támogatja a több felhasználó kezelését, ami fontos, hiszen a szerver alkalmazásom több felhasználó egyidejű kiszolgálására terveztem. Emellett, a egy céloknak megfelelően gyors motorról van szó.

Szemantikus keresés beépített módon nincs támogatva, ezért ezt a rendszeremnek kell biztosítania. Viszont, támogatja a szinonima szótár feltöltését és használatát, illetve tetszőleges formátumú szövegekben lehet keresni, ami lehetőséget nyújt a tudás alapú információ beépítéséhez. Emellett, rendelkezik számos szűrővel, amiket fel lehet használni szövegnormalizálás során, de egyelőre ezt a feladatot inkább az R scriptekre bíztam.

Lucene alapján, Elasticsearch is indexekben tárolja a dokumentumokat. Ezek az indexek adják meg a tárolás struktúráját. Indexeken belül meg lehet határozni az egyes dokumentumok típusát (ami tetszőleges lehet), ezzel tovább bontva a tárolási hierarchiát. Keresést lehet végezni az egész adatstruktúrán, megadott indexen belül, illetve meghatározott típus alapján is.

Flask, Python

Magát a szerver alkalmazást Pythonban írtam, aminek több oka is van. Először is, számos csomag és keretrendszer támogatja a Pythonban írt szerver alkalmazások fejlesztését. Emellett, Python egy népszerű nyelv, ezért a legtöbb szerverként alkalmazható szoftver rendelkezik Python interfésszel, ami megkönnyíti az eszközök rendszerbe való integrálását. Harmadrészt, a Python egy magasszintű nyelv, ami jelentősen megkönnyíti a komplex rendszerek felépítését.

Ez a Python szerver alkalmazás szolgálja ki a bejövő HTTP kéréseket, REST API-n keresztül. A REST API-hoz Flask keretrendszert [9] választottam. A Flask egyik fontos előnye az, hogy alapvetően egy vékony interfész, tehát rugalmasan lehet használni, és nem igényel sok erőforrást. Flask segítségével definiálok és kezelek néhány végpontot, amiken keresztül jönnek a kérések. Ezekre válaszként a keresett és választott dokumentumokat küldi vissza, JSON formátumban.

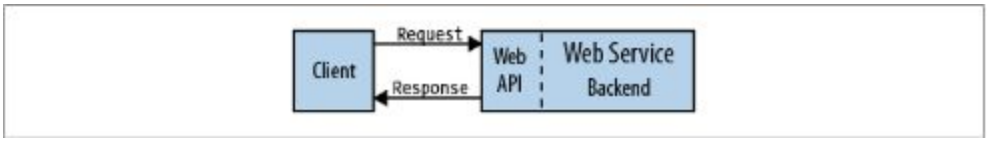
Flask segítségével meg lehet oldani a kérések kiszolgálását dedikált web szerver nélkül, mivel rendelkezik egy beépített web szerverrel, ami természetesen az éles környezetben erősen korlátozott hatékonyságú, viszont a teszteléshez elegendő.

Támogatja a Jinja2 használatát, ami egy templating nyelv Pythonhoz [10]. Ennek segítségével könnyen lehet generálni HTML oldakat. Jelenleg még nem használom, de tervezem beleintegrálni későbbiekben.

REST API

A webes világ szerver-kliens modellre épít [11]. A szerverek és kliensek különböző szerepet töltenek be, és ezek egymástól függetlenül valósíthatóak meg, különböző nyelvekben. Csak annyi feltétel van, hogy be kell tartaniuk a web egységes interfészét.
Ez az egységes interfész a következő tulajdonságokkal rendelkezik:

- Minden erőforrás rendelkezik egy egyedi azonosítóval, ez az URI (uniform resource identifier)
- Kliensek manipulálhatják az erőforrások formátumát
- Kliensek az erőforrások kívánt állapotát közvetítik az üzenetekben. A szerverek ezeket elfogadhatják vagy elutasíthatják, majd az erőforrás jelenlegi állapotát küldik vissza.
- A web hivatkozások által van összekötve.



2. ábra. REST API [11]

A REST API egy megoldást jelent a web szolgáltatások felépítéséhez. A kliensek API-kat (application programming interface) használnak az adatok lekéréshez és azok módosításához. A statikus fájlok kiszolgásának logikáját, az erőforrás modellt követi. URI-k segítségével azonosítja az egyes műveleteket, azonban azok tetszőleges funkciók lehetnek. A paramétereket URI-ban és a HTTP üzenet törzsében adhatjuk meg.

Rserve, pyRserve

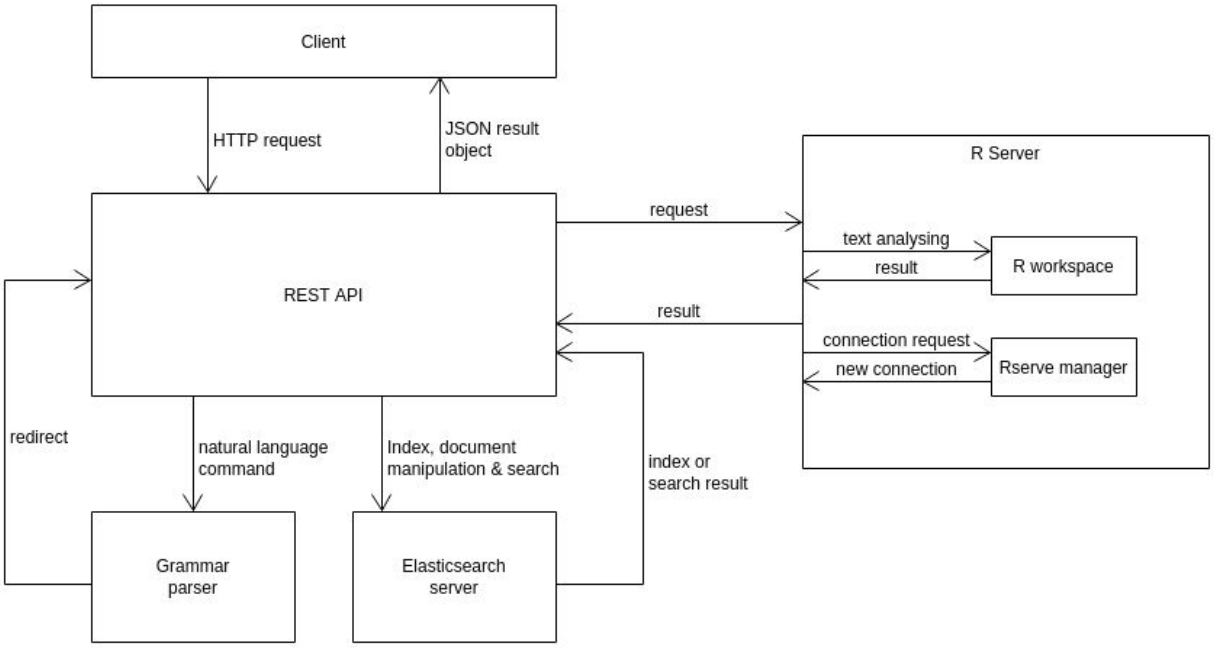
A szövegek elemzéséhez és manipulásához Az R nyelvet vettem igénybe. Ennek az az oka, hogy az számos kiforrott stilisztikai és szövegelemzési csomaggal rendelkezik [6]. Tehát

szükségem van egy R szerverre, ami képes kommunikálni a Flaskos alkalmazással. Erre a célra az Rserve-t használom [12]. Más R szerver implementációkkal ellentétben, az Rserve lehetőséget biztosít különböző munkaterek és kapcsolatok fenntartására, ami lehetővé teszi több felhasználó kiszolgálását.

Emellett Rservehez számos nyelvben van implementált kliens, köztük Pythonban is [12]. Ehhez használom a pyRserve kliens csomagot, ami megkönnyíti az Rserve szerverrel való kapcsolat kezelését.

Összeállított rendszer működése

A rendszer magasszintű működését a 3. ábra illusztrálja:



3. ábra. A rendszer felépítése

Van egy REST API interfész, ami HTTP kéréseket fogad. Csak ez van nyitva a külvilág felé. Az interfész nem végez belső logikát, csak ellenőrzi és kicsomagolja a POST kérések paramétereit, és továbbítja a kéréseket a megfelelő komponensek felé. Végül JSON formátumban küld vissza az eredményeket és az értesítéseket.

Az indexek kezeléséhez és kereséséhez kapcsolódó kéréseket a kereső interfésznek küldi. Az éri el az Elasticsearch futó szerverét. Maga az Elasticsearch szerver pedig lokálisan fut.

A szöveg manipulálására és elemzésére alkalmas műveleket pedig az R szerver végzi, tehát ezeket a kéréseket annak továbbítjuk. Az R szervert Rserve segítségével biztosítom, az is folyamatosan fut lokálisan. Az R munkaterek maguk tárolják a dokumentum állapotát, azt nem szükséges paraméterként folyamatosan küldeni. Minden új felhasználó csatlakozásakor az R szervertől egy újabb munkateret, kapcsolatot kérünk. Így tudunk egyszerre több felhasználót kiszolgálni.

Amennyiben egy természetes nyelvi kérés érkezett, azt a nyelvi elemzőnek küldjük. Az egyszerűen visszatér egy átirányítással egy másik REST végponthoz, amennyiben sikeres volt a parancs feldolgozása. Más esetben hibaüzenettel tér vissza.

Komponensek

Alábbiakban összefoglalom a rendszerem fontosabb komponenseit és azok működését. Itt már a komponensek egymással való kommunikációját és a belső működését mutatom be.

REST API

Mint ahogy azt korábban említettem, a REST API-s kérések kiszolgálásához és végpontok biztosításához a Flask keretrendszert használom. A végpontok struktúrája alapvetően a következőképpen néz ki:

- */parancs/<index>*

Itt a parancs az az Elasticsearch index manipulálás, keresés, szövegelemzés lehet. Az index változó pedig azt jelzi, hogy melyik indexen akarunk végezni műveletet. Nem minden művelethez tartozik index, természetesen ilyenkor azt nem adjuk meg. Itt van néhány példa az indexekkel rendelkező végpontokra:

- */create/<index>*
- */delete/<index>*
- */upload/document/<index>*

Az első és második példa értelemszerűen létrehozza, illetve törli a megadott indexet. Index törlésénél minden dokumentum törlődik, ami az adott indexhez tartozik. A harmadik esetben pedig feltölthetünk egy dokumentum fájlt a megadott indexbe. Az utóbbi végpontnak az az érdekessége, hogy amennyiben *GET* típusú kérést küldünk, akkor kapunk egy HTML sémát a fájl feltöltéséhez. A séma tartalmaz egy formot, amiben ki lehet választani egy fájlt és *POST* kéréssel elküldeni. Az utóbbi formában kapott kéréseket pedig úgy dolgozzuk fel, hogy a fájl tartalmát kiolvassuk és azt feltöltjük az Elasticsearch szerverre egy JSON objektumként. A kérés *GET* változata igazából egy kényelmi funkció ahhoz, hogy böngészőben is meg lehessen nyitni és elküldeni a fájlt.

Az összes, Elasticsearch szerveréhez kapcsolódó funkciót annak továbbítjuk, és az elvégzi a műveleteket. A dokumentumok elemzéséhez és manipulálásához tartozó kéréseket pedig az R szervernek küldjük. A kérések megfelelő módon történő továbbítása után az eredményt JSON formátumban küldjük vissza a kliensnek.

Van még egy különleges végpont, ez pedig a természetes nyelvi interfészhez tartozó csomópont, amit a következő URL-en éjük el:

- */parse/<command>*

A parancs feldolgozását a parsernek küldjük el, amely feldolgozza a kérést és egyszerűen átirányít a megfelelő végpontra. Amennyiben a parser nem tudta értelmezni a parancsot, akkor egy hibaüzenet formájában értesítjük a klienst.

Természetes nyelvi interfész

A nyelvi elemzőt nem manuálisan írtam, hanem generáltam egy saját alkotású parser generátorral. A parser generátorról egy külön részben írok részletesen, itt a már generált nyelvtanról és annak felhasználásáról írok.

A generált nyelvi elemző Earley algoritmust használja az elemzéshez. Earley algoritmus egy *chart parser* implementációja, erre a generátort leíró részben térek ki részletesebben.

Mint említettem, a nyelvtan elemző egy speciális REST végponttól kapja a bemenetet, ami egy kontrollált természetes nyelvi parancs. A parancs feldolgozása után egyszerűen átirányítja a megfelelő REST API végpontra, és feltölti adatokkal, amennyiben egy POST kérésről van szó.

A nyelvtan struktúrája ilyen módon néz ki:

Vannan tokenek, amik tartalmazzák az összes kulcsszót és azok szinonimáit. Például, a keresés parancsát a következő kifejezésekkel tudjuk elkezdeni:

- Search for, look up, find

Ezek a kifejezések mind egyenértékűek, utánuk pedig egy tetszőleges szót várunk, ami a keresés kulcsszava lesz. Következő lépésben megadhatjuk azt, hogy milyen indexben szeretnénk keresni. Amennyiben azt nem jelöljük meg, a kereső az összes indexben fog keresni. Íme néhány példa az érvényes kérésekre:

1. Create index: docs1
2. Upload document to: docs1
3. Search for: kék in index: docs1

A parancsok értelmében, a felhasználó először létrehoz egy *docs1* nevű indexet. Ha ilyen index már létezik, akkor arról kapunk egy értesítést (ezt a logikát már nem az elemző végzi, az csak átirányít a megfelelő függvényekre). Látszik, hogy az *index:* kifejezés fogja felkészíteni az elemzőt arra, hogy egy tetszőleges elnevezés jön. A nyelvtan megírásánál figyelni kell arra, hogy legyenek egyértelmű részei, még ha az egész nyelvtan nem is egyértelmű. Az egész nyelvtan egyértelműsége azért nem követelmény, mert a chart parser a nem egyértelmű környezetfüggetlen nyelvtanokat is képes elemezni. Ez azért lehet hasznos, mert a nem egyértelmű kéréseknél jelezni tudjuk a felhasználónak az összes lehetőséget, amire illeszkedhet a kérése, így következő alkalommal pontosabban tud fogalmazni.

Következő lépésben a felhasználó fel szeretne tölteni egy dokumentumot a *docs1* indexbe. Ilyenkor egy átirányítással megjelenítjük neki a fájl feltöltő oldalt. A fájl tartalma pedig a megadott indexbe kerül, mivel átirányítás során az elemző azt paraméterként átadta. A mostani példában a felhasználó kihagyta az *index* kulcsszót, ami nem probléma, mert az egy *opcionális*

token. Valójában a *kettős pont szimbólum* fogja jelezni a tetszőleges, whitespace és speciális karakter nélküli karakterlánc kezdetét.

Végül a felhasználó egy keresést végez: a *kék* kulcsszóra keres rá. Ezt a kérést megkapja az Elasticsearch és visszaadja az összes találatot.

A nyelvtan bemutatásához az alábbiakban leírom a megoldásommal izomorf struktúrát EBNF formában, ami az index létrehozását teszi lehetővé. A nyelvtan leírását 1. táblázatban láthatjuk.

create_index = create_term, get_index
create_term = “make” | “create” | “start”
get_index = [“index”], colons, term
colons = “:”
term = "[a-zA-Z0-9_]+"

1. táblázat. Kiterjesztett ENBF példa

Ez a rövid leírás szemlélteti, hogy milyen módon írhatjuk le az index létrehozását. *Create_indexhez* két, egymás utáni terminusra van szükségünk: *create_term* és *get_index*.

Create_term pedig három különböző tokenből állhat: “make”, “create” vagy “start”. EBNF leírásban ‘|’ szimbólum a “vagy” kapcsolatot jelzi. A szögletes zárójelek közötti részben pedig egy opcionális kifejezés van. Látszik, hogy a termnél egy reguláris kifejezést használtam, ami EBNF értelmében nem megengedett, de egyrészt így tömörebb a kifejezése, másrészt az elemző generátorom is megengedi. Tehát itt csupán azt írtam le, hogy az elnevezés egy tetszőleges, egynél nagyobb számú betűből, számból és alulvonás karakterből állhat.

Hasonló módon van megfogalmazva a többi megengedett bemenet, és ezek együttesen leírhatóak EBNF formában. A nyelvtan különválasztása a szerver logikájától pedig jobb rugalmasságot és bővíthetőséget biztosít.

R szerver

Az R szerverem két részből áll. Egyrészt, van a rserve manager: az ő dolga a különböző felhasználók számára kiosztani R kapcsolatokat. Ez egy folyamatosan futó szerver alkalmazás, amitől a fő alkalmazás kér kapcsolatok létrehozását, amikor egy újabb felhasználó akar csatlakozni.

Ezen kívül, minden kapcsolathoz tartozik egy munkatér. Python interfészen keresztül végre lehet hajtani natív R scripteket. Így két lehetőséget is biztosítom R függvények alkalmazásához: elsődlegesen, a legfontosabb funkciókhoz rendeltem Python függvényeket, amik a paramétereket átadva, előre definiált R függvényeket futtatják. Ezeket a természetes nyelvi interfészen és a REST API-n keresztül el tudjuk érni. Ezek a függvények folyamatos bővítése egy konstans fejlesztési lehetőséget ad az alkalmazás számára.

Viszont, amennyiben egy jártasabb felhasználóról van szó, aki ismeri az R nyelvet, neki lehetősége van tetszőleges R kód végrehajtásához.

Az R függvények által létrehozott módosításokat a szerver nem menti el automatikusan, helyette mindig a korpusz (R-ben így hívják a dokumentumokat összességét) jelenlegi állapotát tárolja. Ezt azonban el lehet menteni Elasticsearch szerverére.

Hatékonyság növelése érdekében, a korpusz friss állapotát eltároljuk az R munkatérben, azt nem Python objektumból kapja paraméterként. Ezért van egy olyan függvény, ami a megadott indexhez tartozó dokumentumokat összecsomagolja és azokból készít egy korpuszt, amit átadja az R interfésznek. Így, jelenleg az R műveleteket csak egy adott indexhez tartozó dokumentumokon lehet elvégezni.

Amennyiben a felhasználó feltölt egy szótárat, azt az R interfész is megkapja. Ezt a szótárat lehet alkalmazni a szöveg normalizálásához (pontosabban egy olyan művelet, amit angolul *stemming*nek hívják). Stemming egy beépített funkció az R SnowballC nevű csomagjában: a megadott nyelv alapján az adott szövegben minden szót visszavezeti szótári alakjára. Általában meg lehet adni, hogy milyen agresszívan végezze a műveletet. Ez egy érdekes dilemmához vezet. Vegyük a következő példát. Adott néhány angol szó:

- walk, walks, walked, walking, walker, walkers

Stemming elvégzése után a következő eredményt kapjuk:

- walk, walk, walk, walk, walker, walker

Ez volt a várható eredmény, és ez teljes mértékben megfelel. Itt valójában két műveletet is végzünk: először vágunk le a szavakból, aminek az eredménye sokszor egy nem létező szótő. Következő lépésben pedig stem completion műveletet végzünk, ami az eredeti szöveg alapján megpróbálja kitalálni a legjobban illeszkedő szóalakot.

Azonban, a magyar nyelvű, főleg régies kifejezésekkel teli szövegre sokkal rosszabb eredményt kapunk. Tekintsük a következő példát:

- Amerika, America, Americában, amérika, amérika, amérikában, ámérika, ámérikában

Ezeket az szóalakokat ma már nem használjuk, viszont a régi irodalmi szövegekben előfordulnak, és így például a Mikes Kelemen Törökországi levelek művében is megtalálhatóak. Ha ezekre a példa adatokra futtatjuk az R SnowballC csomag beépített stemmerjét, akkor az alábbi eredményt kapjuk:

- America, America, America, amérika, amérika, amérika, ámérika, ámérika

Látszik, hogy sajnos nem a megfelelő megoldást kaptuk. Ezért az alkalmazásomban arra mutatok példát, hogy saját szótár használatával hogyan érhetünk el jobb eredményt.

Én egy egyszerű algoritmust implementáltam a szótár alapú szöveg normalizálásához, ami sajnos nagy szövegek mellett hosszabb futási időt igényel. Viszont az eredmény demonstrálásához ez is elegendő. A következőképpen működik:

Bemenetként vár egy szavak listájából álló listát. Az egyes szó listák első helyén álló kifejezés mutatja azt, hogy milyen alakra kell visszavezetni a soron következő szóalakokat. Tehát annyi szólistát várunk, ahány szótári alakú szóval rendelkezünk.

Az algoritmus egyszerűen végigmegy a megadott szöveg minden egyes szaván, és megvizsgálja, hogy ahhoz a szóhoz milyen szótári alak tartozik és arra cseréli le. Ez egy mohó algoritmus, aminek a futási idejét a szótár mérete és a szöveg hossza határozza meg. A beépített R stemming algoritmusok ennél jelentősen hatékonyabban működnek, de azok sajnos

nem tudnak alkalmazni tetszőleges szótárakat. Az algoritmusom futtatásának eredményét később, az utolsó részben mutatom be.

A szótárat egy egyszerű csv formátumú fájlként lehet feltölteni, ahol a különböző szavak soronként szerepelnek, a szavakon belüli eltérő szóalakok pedig vesszővel vannak elválasztva. Ezt az egyik REST API végponton lehet feltölteni, amit utána egyaránt az R interfész és az Elasticsearch is megkapja.

Végül lehetőség van a szövegelemzéseknél gyakran használt *szófelhő* generálásához. Ez egy szemléletes módon illusztrálja az adott szövegek leggyakrabban előforduló szavait. Ez egy hasznos eszköz lesz az eredmények ábrázolásához. Egyelőre ez a művelet fix paraméterekkel dolgozik, de jövőben ezt is rugalmasabbá lehet tenni.

Kereső

A kereső interfésze az alapvető Elastissearch funkciókat biztosítja a felhasználó számára. Ezek pedig a következők: indexek manipulálása, fájlok és szótárak feltöltése, illetve a keresés.

A kereséshez alapvetően két mód van: kilistázni az összes dokumentumot (adott vagy minden indexből), illetve rákeresni egy adott kulcsszóra. Amennyiben egy adott kulcsszóra keresünk rá, megkapjuk azokat a dokumentumokat, ahol a kulcsszó szerepelt, azok metaadatait (melyik index, a dokumentum címe) és a találatok kiemelését. Az utóbbiból könnyen készíthetünk snippeteket, amit a webes front-enden felhasználhatjuk. Jelenleg az összes eredményt egy JSON objektum formájában kapjuk.

Amennyiben az indexre beállítunk egy szinonimaszótárat, a keresés során nem csak az adott kulcsszót, hanem annak szinonimáit is keresni fogja.

Ontológia kezelő

Ahhoz, hogy mélyebb szemantikus információt biztosítsunk a nyers szövegeknek, szükség van egy ontológia kezelőre. Legegyszerűbb esetben az előre megadott kategóriák szerint lehet dolgozni (például tulajdon nevek, helységek). A feltöltött RDF fájlt feldolgozza, minden megadott kategória alapján kigyűjti az RDF-ben talált kifejezéseket, és minden entitást külön fájlba menti. Az RDF fájlt is megindexeli, hogy keresést lehessen végezni rajta.

Az egyes entitásokra keresést futtatunk a megadott indexben, és megtaláljuk az összes dokumentumot, ahol az szerepel. Az adott entitások tartalmaznak egy listát, hogy mely dokumentumokban találhatóak. Így, amikor keresést végzünk az adott indexben, akkor nem csak a dokumentumokban keressük a megadott kulcsszót, hanem az RDF indexben is, és az alapján meg tudjuk határozni, hogy az kapcsolódik-e az adott dokumentumhoz.

Ez a komponens még nincs megvalósítva a jelenlegi rendszeremben, de a rendszer struktúrája megengedi az előbb bemutatott terv szerinti implementációt.

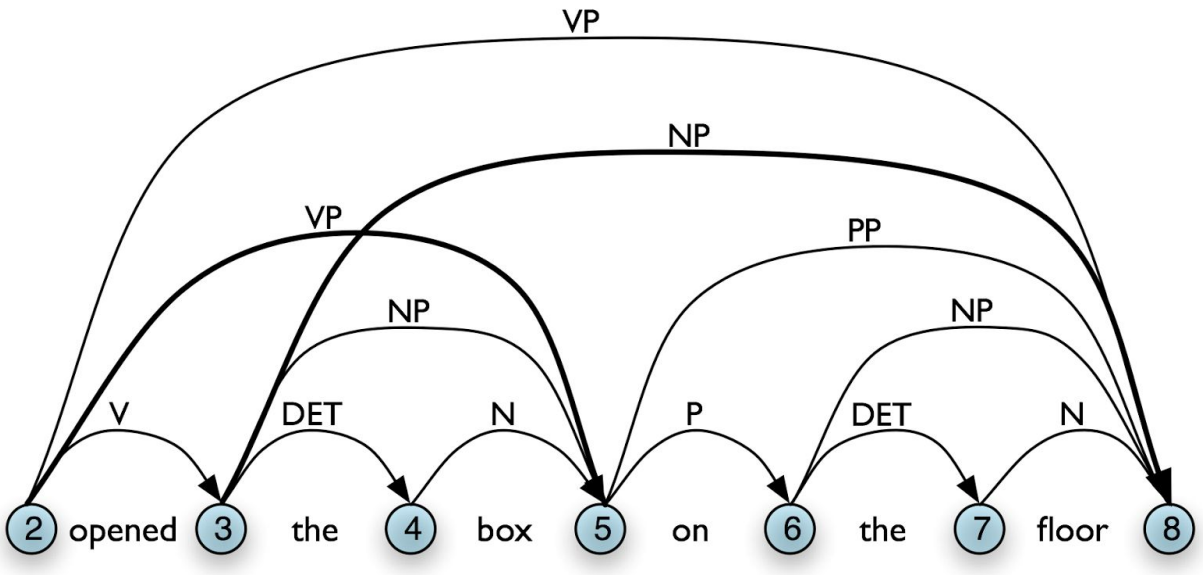
Nyelvtan parser generátor

Írtam egy saját nyelvtan parser generálót Pythonban. A generátorom különlegessége az, hogy egy chart parsert generál [14], míg általában a parser generálók általában LL(k) elemzőket készítenek. Az utóbbi előnye az, hogy gyorsabb a futása, viszont sokkal korlátozottabb a nyelvtan. Csak egyértelmű környezetfüggetlen nyelvtanokat enged meg, ezen belül is a *k* értéktől függően tovább korlátozza a nyelvtant. Lényegében ez az érték azt mutatja, hogy hány (k darab) tokent nézi meg előre az elemző, hogy eldöntse milyen szabályra illeszkedik a megadott bemenet.

Ezzel szemben a chart parser tetszőleges környezetfüggetlen nyelvtanokat enged meg, még a nem egyértelműeket is. Az utóbbi eset hasznos lehet, ha felhasználónak jelezni szeretnénk, hogy milyen módon lehet interpretálni a bemenetét, így következő alkalommal pontosíthatja azt.

Az általánosabb nyelvi elemzőmhöz futási időben lehet rendelni nyelvtani szabályokat. Ez az implementáció köré építettem egy genetátort, ami a metanyelvtan szabályai szerint egy Python scriptet generál, ami a kész elemzőt adja meg.

Chart parser



4. ábra. Chart parser példa [14]

A chart parser dinamikus programozás elvén alapul. Ebben az esetben azt jelenti, hogy szavakra bontja a problémát, és megpróbál lokálisan megoldani a feladatot aztán az eredményeket összerakni, hogy egy egész mondat elemzését megkaphassuk. Ez úgy néz ki, hogy minden szónál eltároljuk, hogy az milyen nyelvtani szabályra tudna illeszkedni a bemenet alapján, aztán próbálunk egyrészt jósolni, hogy milyen szabályra fog illeszkedni, másrészt a következő bemeneti szó alapján eldönteni, hogy ezek közül melyik szabályra tud illeszkedni.

A parsernek három fő komponense van: predictor, megpróbál jósolni, hogy milyen szabály következik. Ilyenkor egyszerűen átírja a nyelvtani szabályok bal oldalán lévő változót, helyettesíti a jobb oldallal. Minden szabályt átírja, aztán jön a scanner működése. Scanner egyszerűen beolvassa a következő tokent (a szövegben lévő termináló szót). Ezek bekerülnek a következő szóhoz tartozó táblázatba. Végül completer fogja egyesíteni a részmegoldásokat: megvizsgálja, hogy a predictor által létrehozott szabályok közül melyik „várja” a befejezett szabályt és ez alapján egyesíti azokat.

Ahhoz, hogy a végén vissza lehessen követni, hogy az egyes szabályok mely részekből álltak össze, a completer még elmenti azok referenciát.

Earley algoritmus implementálása

Először is bemutatom az általánosabb Earley algoritmus [15] implementációmat. Ezt alacsony szinten lehet kezelni, futási időben hozzáadni a nyelvtanhoz új szabályokat.

Alapvetően két komponensből áll. Egyrészt, van a nyelvtan objektum. Ez a nyelvtan a megszokott környezetfüggetlen nyelvtanokat reprezentálja. Tehát a nyelvtan átíró szabályokból, terminális szimbólumokból és változókból, más szóval, nemterminális szimbólumokból áll.

A terminális szimbólumok általában egyszerű karakterláncokat jelentik, viszont az én nyelvtanom reguláris kifejezéseket is megenged. Ez sokszor megkönnyíti a nyelvtan definiálását.

Az átíró szabályok pedig egyes változóhoz rendelnek változókat és terminális szimbólumokat. Ugyanahhoz a változóhoz több szabályt is lehet rendelni. Azt mondjuk, hogy az adott bemenetet az adott nyelvtan akkor fogadja el, ha a kezdeti változó és átíró szabályokkal le tudjuk képezni a bemeneti szót úgy, hogy a végén már csak terminális szimbólumok maradnak. Tehát mindenképpen van egy kezdeti változó is.

A nyelvtanom egy objektum, ami egy halmazban tárolja a szabályokat és a lehetséges terminálisokat. A nyelvtanhoz akármikor hozzá lehet adni új szabályokat egy függvény segítségével. Ezt a nyelvtan objektumokat fogja használni az Earley algoritmus.

Tehát a másik komponensem az Earley algoritmust implementálja. A bemenetet először tokenekre bontja az alapján, hogy milyen terminális szimbólumok vannak a nyelvtanban. Ez után inicializálja az állapotokat. Az állapotokat a chart parser működése szerint változtatja.

Végül, az utolsó táblázatban lévő állapotokat végigvizsgálja, és azok, amelyekhez a kezdeti változó alapján tudtunk eljutni, azok lesznek a lehetséges megoldások. Ezekből épít fa struktúrákat és egy listában fogja visszaadni az algoritmust elindító függvényben. Ez a fa struktúra tárolja az egyes szavakat és azokat a szabályokat, amik segítségével megkaptuk azokat. Ezt a fa struktúrát lehet majd felhasználni a szemantikus jelentés hozzárendeléséhez. Annyit fát hoz létre, ahányféleképpen lehetett értelmezni a bemenetet.

A metanyelvtan

A generátorom működése leginkább JavaCC parser generátorhoz hasonlít. JavaCC [16] a legnépszerűbb parser generátor Javahoz. Onnan merítettem ötleteket a metanyelvtan megalkotásához.

A feladat tehát, egy fájlban megadott, kiterjesztett EBNF formátumú nyelvtan alapján generálni elemzőt. Ehhez definiáltam egy saját nyelvtant, ami leírja ezek a generálandó nyelvtanokat. A metanyelvtanom EBNF-re épít, de lehetőség van egyrészt reguláris kifejezések használatára a terminális szimbólumok megadásánál, másrészt szemantikus jelentést is hozzá lehet rendelni az egyes átíró szabályokhoz. A generátorom megalkotása közben az volt a célom, hogy a szintaxist és a szemantikát össze lehessen fűzni. Ezt úgy értem el, hogy az egyes átíró szabályokat függvények formájában lehet megadni, és azokhoz lehet Python kódot írni, amit a generált elemzők végre fogják hajtani a megfelelő átíró szabályok függvényében.

A nyelvtanok elemzéséhez már nem chart parsert alkalmazom, hanem előre definiált függvényekkel, amik egy LL(1) parsert valósítanak meg. Mivel a formális nyelvet elég egyszer jól definiálni, így nincs szükség komplex nyelvi elemzőre.

A nyelvtanok definiálását egy egyszerű példán keresztül mutatom be, amit az 5. ábrán tekinthetjük meg:

```
# Example: algebric expression evaluator

Num: "[0-9]+"
OpExpr: '+' | '-'
OpProduct: '*' | '/'
SKIP: '\s'

!Formula():
    expansion:
        result = Expr()
    end:
    {
        print(result)
    }

Expr():
    begin: {
        a = 0
        b = 0
        op = '+'
    }
    expansion:
        a = Term() |
        a = Expr(), op = OpExpr, b = Term()
    end:
    {
        if op == '+':
            return a + b
        else:
            return a - b
    }

Term():
    begin: {
        a = 0
        b = 1
        op = '*'
    }
    expansion:
        a = Factor() |
        a = Term(), op = OpProduct, b = Factor()
    end: {
        if op == '*':
            return a * b
        else:
            return a / b
    }

Factor():
    expansion:
        a = Number() |
        '(', a = Expr(), ')'
    end: {
        return a
    }

Number():
    expansion:
        num = Num
    end: {
        return int(num)
    }
```

5. ábra. Nyelvtan definiálása

Először meg kell adni a tokeneket, ezek lesznek a terminális szimbólumok. Ezek alapján fogjuk tokenekre bontani a bemenetet, illetve ezekre lehet hivatkozni az átíró szabályokban. Ebben az esetben, a bemenetet számokra és operátorokra bontjuk. A *SKIP* kulcsszó pedig egy speciális tokent jelenti: ezeket a nyelvi elemző ki fogja hagyni token generálás során. Így, a jelenlegi példában a whitespace karaktereket ignorálni fogja, ezért az alábbi két bemenet ekvivalens lesz:

- 2+3*4
- 2 + 3 * 4

Egyes tokenekhez “|” szimbólummal lehet több terminálist rendelni. Ezek ekvivalens szavak lesznek a token szempontjából.

Terminálisokat kétféleképpen tudunk definiálni: aposztrófok között egy egyszerű karakterláncot lehet megadni, idézőjelek között pedig egy reguláris kifejezést írhatunk.

A változókat függvényekhez hasonlóan definálhatunk. Kaphatnak paramétert, és így, amikor használunk egy változót, akkor a definiált paramétereket átadhatjuk. Van egy speciális változó, azt jelöljük a felkiáltójellel. Ez lesz a kezdeti változó.

A változók három részből állnak: van egy *begin* rész, kapcsos zárójelek között megadhatjuk azt a Python kódot, amit lefuttatjuk minden alkalommal, amikor ezt a változót kapjuk. Hasonlóan, az *end* részhez azt a kódot írjuk, ami a függvény végén fut le. Ide általában valami visszatérési érték kerül, hogy a változókból eredményeket lehessen kinyerni.

A középső rész az az *expansion*: itt meg lehet adni az átíró szabályokat. A különböző átíró szabályokat “|” szimbólummal lehet jelezni. Ezeket mind ugyanahhoz a változóhoz fogjuk rendelni.

Szabályokon belül pedig vesszővel elválasztva írhatjuk le, hogy milyen tokenekre (amik lehet változók és terminális szimbólumok egyaránt) lehet átírni azokat. A terminális szimbólumok használatakor lehetőségünk van a már előre megnevezett, definiált tokenekre hivatkozni, de itt helyben is lehet definiálni.

Ahogy a példában is látszik, a változókat és a terminális tokeneket változókba lehet elmenteni. Ennek eredményesképpen egy string formájában kapjuk az illesztett tokent. Mivel ebből Python kódot generálunk a végén, ezeket a *begin* és *end* részben tudjuk használni.

Mint látszik, a fent megadott nyelvtan egy egyszerű algebrai kifejezés kiértékelőt írja le, ami ismeri az összeadás, kivonás, szorzás és osztás műveletét. Mellesleg a zárójeleket is tudja kezelni. Általában, kézzel, egy ilyen elemző megírása nem egy rendkívül nehéz feladat, de azoknak problémát okoz, akik még nem foglalkoztak nyelvi elemzőkkel. Ráadással, ennél sokkal hosszabb módon kellene leírni azt, és kevésbé áttekinthetően. A nyelvtan módosítása pedig általában az egész program jelentős részének módosítását jelenti, itt pedig csak a nyelvtant leíró fájlt kell módosítani.

Elemzők generálása

Mint ahogy azt korábban említettem, az Earley algoritmus által használt nyelvtan objektumomhoz egyszerű függvényhívásokkal lehet hozzárendelni új szabályokat. Ezt használom ki az elemző generálásához.

Egy Python scriptet írok a generálás során. Először létrehozom a nyelvtan objektumot, aztán azt feltöltöm terminális szimbólumokkal. A változók feldolgozása pedig a következő módon zajlik: létrehozok változókat és azokhoz tartozó átíró szabályokat. Ezeket megkapja a nyelvtan. Mint, ahogy a metanyelvtan formájából sejthetjük, a változókhoz tartozó törzsből Python függvények lesznek: ezek eleje a *begin* rész, utána az átíró szabályokból változók és függvényhívások lesznek, a végén pedig az *end* rész lesz.

Mivel egy változóhoz több átíró szabály tartozhat, ezek mind külön függvények lesznek, a nyelvtanban külön szabályoknak minősülnek. A szabályokhoz hozzárendelem ezeket a generált függvényeket.

Végül az elemző futtatása két részben zajlik: először maga az Earley algoritmus elkészíti az absztrakt fákat, amik leírják az adott bemenetet. Ezeket egy listában visszaadja.

Ezek után a külső alkalmazásban végig lehet iterálni a fa objektumokon és bejárni azokat. A fa bejárása során az alkalmazott átíró szabályoknak megfelelően hívjuk a generált függvényeket, így a fa bejárása automatikusan a hozzárendelt szemantika végrehajtását jelenti.

Eredmények

Alábbiakban bemutatom a rendszerem tesztelése során keletkezett eredményeket. Látszik, hogy javul a transzformáció és keresés eredménye, amennyiben szótárat alkalmazunk.

Minta adatok

A dokumentum, amivel itt dolgoztam, az a Mikes Kelemen Törökországi levelek nevezetű műve. Ez a minta jól illusztrálja, hogy a régies szövegek feldolgozásának hatásfokát milyen mértékben lehet növelni szótár alkalmazásával. A dokumentum egy nyers szöveg, txt formátumban van megadva. 6. ábrán láthatunk egy részletet a dokumentumból.

```
20   Gallipolibol. Aö. 1717. 10. 8bris.
21
22   Édes néném hálá légyen az Istennek, mi ide érkeztünk ma szerencsésen, franczia országbol
     pedig 15 7bris indultunk meg. a fejdelmünknek Istennek hálá jo egéssége volna. hogy ha a
     köszvény bucsut akarna tölle venni. de reméllyük hogy itt a török áer el üzi. édes néném
     mi jó a földön járni, láttya kéd még sz. péter is meg ijedet volt mikor avizben
     sipadoztak alábai. hát mi bünösök. hogy ne félnénk amidön a hajonk ollyan nagy habok
     közöt fordult egyik oldalárol, amásikára mint az erdélyi nagy hegyek., némelykor azoknak
     atetején mentünk el, némelykor pedig ollyan nagy völgyben estünk, hogy már csak azt
     vartuk. hogy reánk omollyanak azok a viz hegyek; de még is ollyan emberségesek voltanak.
     hogy többet nem adtak innunk, mint sem kellet volna. elég ahogy itt vagyunk egésségben,
     mert atengeren is meg betegszik az ember, nem csak aföldön. és ót ha a hintó meg ráza, el
     fárad, és job egyepegyéje vagyon az ételre, de a hajoban. aza szüntelen valo, rengetés,
     hánkodás. a föt el bodittya. a gyomrot. fel keveri. és ugy kel tenni, valamint arészeg
     embernek. aki abórt meg nem emésztheti. a szegény gyomromnak is ollyan nyavalyában
     kelletet lenni vagy két elsö nap. de azután ugy kellet ennem valamint a farkasnak. a
     fejdelmünk ahajobol még nem szállot vala le. hogy egy tatár hám. aki itt exiliumban
     vagyon, holmi ajándékot külde és atöbbi közöt egy szép lovat nyergelve. itt a fejdelemnek
     jó szállást adtak, de mi ebül vagyunk szálva. de még is jobban szeretem itt lenni mint
     sem ahajoban., édes néném akéd kedves levelit vagyon már két esztendeje hogy vettem.
     igazat mondok. hogy ha az esztendö egy holnapbol állana. reménlem édes néném hogy már
     ezután mint hogy egy áerrel élünk, gyakrabban veszem kedves levelét. de mint hogy egy
     nehány száz mély földel közeleb vagyunk egymáshoz. ugy tettzik hogy már inkáb is kel
     kédnek engemet szeretni, én pedig ha igen szeretem is kédet de többet nem irhatok. mert
     ugy tettzik mint ha aház keringene velem. mint ha most is ahajoban volnék.
```

6. ábra. Egy részlet Mikes Kelemen: Törökországi levelekből

A tesztelés során ehhez a dokumentumhoz egy *test* nevű indexet hoztam létre. Oda töltöttem fel a szöveget, utána pedig keresést és szövegelemzést végeztem. Először szótár nélkül, későbbiekben szótár segítségével.

A szöveghez adott egy szótár, XML formátumban. Ez tartalmazza a műben előforduló összes szó minden szóalakját, amikhez rendel 1-1 szótári alakot is. A szavak mellett fel van tüntetve az egyes szóalakok előfordulásának száma. A 7. ábrán láthatunk egy részletet a szótárból.

```
29053   <entry headword="csillámló">
29054   <U>csillámló</U>
29055   <Q>tsillámlo</Q>
29056   <B>tsillámlo</B> - 1
29057   <Q>tsillámló</Q>
29058   <B>tsillámló</B> - 1
29059   </entry>
29060
29061   <entry headword="csinogat">
29062   <U>csinogat</U>
29063   <Q>tsimogat</Q>
29064   <B>tsimogattyák</B> - 1
29065   </entry>
29066
29067   <entry headword="csinogatni">
29068   <U>csinogatni</U>
29069   <KT />
29070   <Q>tsinogatni</Q>
29071   <B>tsinogatni</B> - 1
29072   </entry>
29073
29074   <entry headword="csinogatás">
29075   <U>csinogatás</U><L />
29076   <Q>csinogatás</Q>
29077   <B>csinogatásrol</B> - 1
29078   </entry>
```

7. ábra. Teszteléshez használt szótár minta

Mivel az eredeti szótár XML formátumú, az én rendszerem pedig egyszerű csv fájlokat vár, ezért ezt a szótárat még előtte át kellett konvertálnom a megfelelő formátumba.

Keresés

A keresést az Amerika kulcsszó alapján végeztem. A 8. ábrán látszik, hogy a mai formában ez a név egyszer sem szerepel a műben, régies formában pedig többször is.

```
812   <entry headword="Amerika">
813   <U>Amerika</U>
814   <T_/>
815   <F_/>
816   <Q>America</Q>
817   <B>Americában</B> – 1
818   <Q>amérika</Q>
819   <B>amérika</B> – 1
820   <B>amérikában</B> – 4
821   <Q>ámerika</Q>
822   <B>ámerikában</B> – 2
823   <B>ámerikából</B> – 1
824   <Q>améríka</Q>
825   <B>améríkában</B> – 1
826   </entry>
```

8. ábra. A szótárban előforduló szóalakok “Amerika” kifejezésre

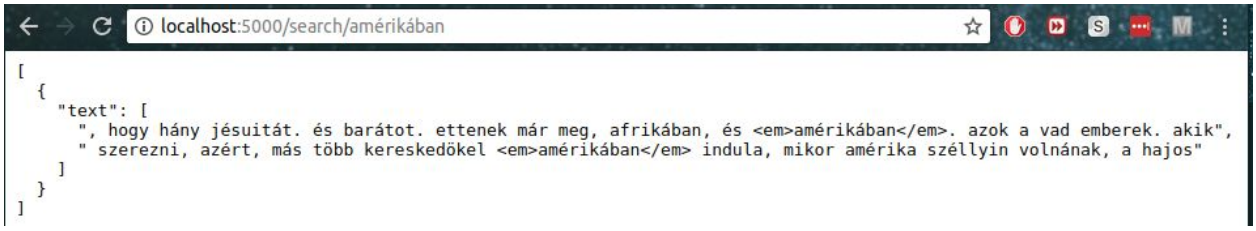
A demonstráció érdekében most csak a kiemelt részeket jelenítettem meg.

Szótár nélkül

Először rákerestem az eredeti műben az “Amerika” kulcsszó alapján. 9. ábrán látszik az eredmény:

9. ábra. Szótár feltöltése előtti eredmény “Amerika” kulcsszó alapján

Nem talált egyet sem, pedig ez a szó szerepel a szövegben, csak más szóalakban. Nézzünk egy példát arra, hogy mi történik, amikor valamelyik régies formára keresünk rá. Ezt a 10. ábra mutatja be:



10. ábra. Szótár feltöltése előtti eredmény “amérikában” kulcsszó alapján

Ilyen szóalakot már megtalálja, sőt, ki is emeli két helyen.

Szótár segítségével

Miután feltöltöttem a szótárat és újra rákerestem az Amerika kulcsszóra, modern formájában, az elvárt eredményt kaptam. Ez látszik a 11. ábrán.

11. ábra. Szótár feltöltése utáni eredmény “Amerika” kulcsszó alapján

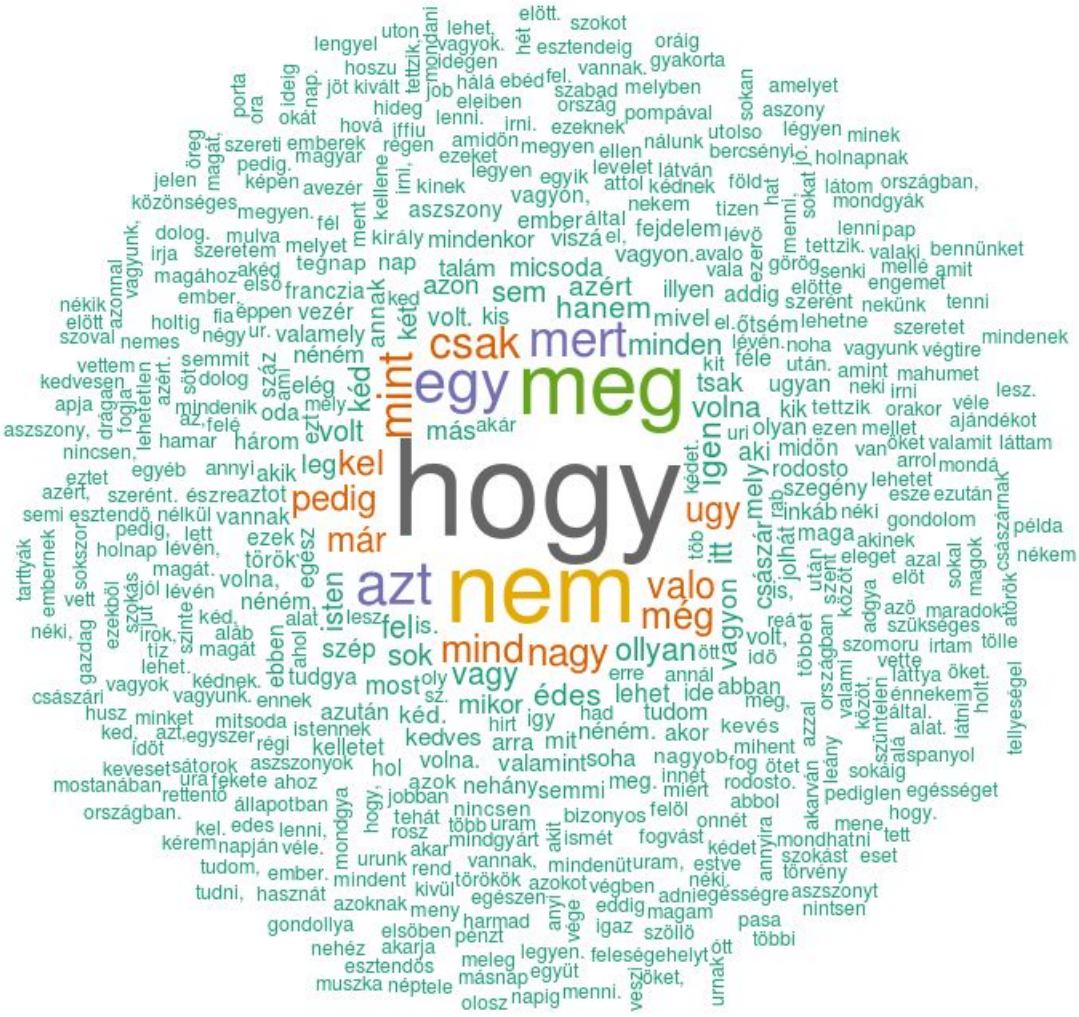
Mivel nem módosítottam az eredeti szöveget, csak egy szinonima szótárat alkalmaztam, ezért a kereső meghagyta a régi szóalakot. Viszont, most már megtalálja azokat a modern szótári alak alapján, illetve ugyanúgy elvégzi a kiemelést.

Szöveg transzformációk

Mint ahogy azt az R szerver leírásában említettem, van egy szó felhő eszköz, ami megjeleníti a szövegben található releváns szavakat. A megjelent szavak mérete az előfordulásaik számával korrelál. Először megmutatom, hogy hogy néz ki ez a szó felhő, ha az egyszerű, nyers szövegre alkalmazzuk.

Transzformáció nélkül

12. ábra alapján látszik, hogy mi eredmény, ha nem hajtunk végre semmilyen szöveg transzformációt.



12. ábra. Nyers szövegre alkalmazott szófelhő eredménye

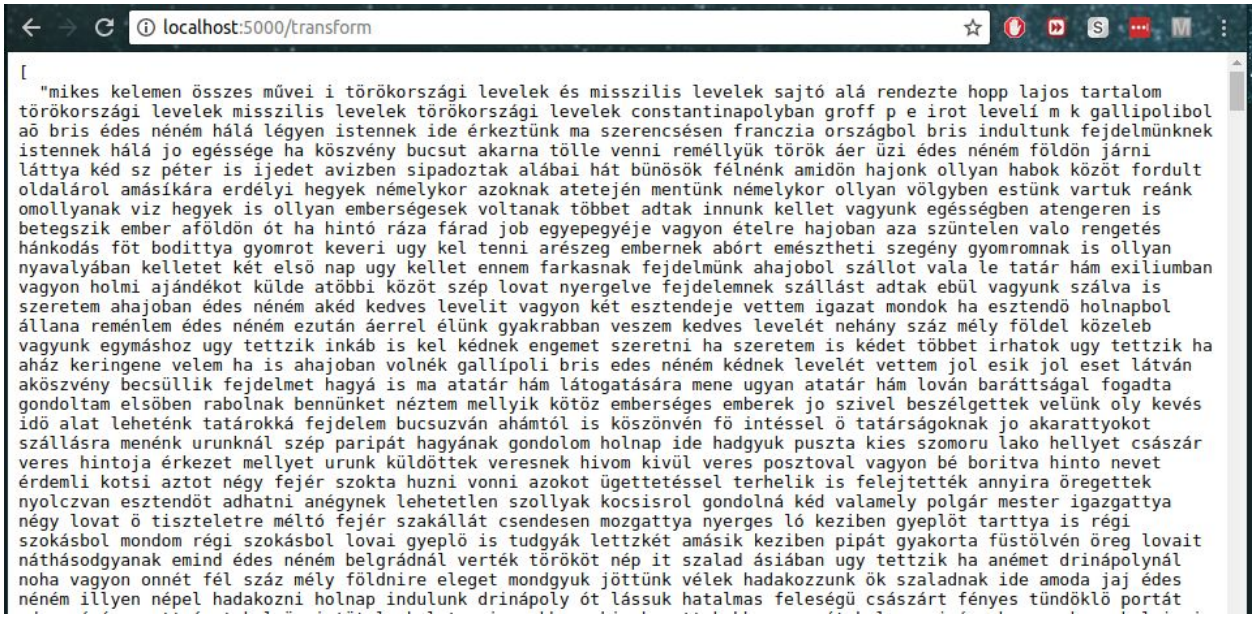
Ezzel az eredménnyel több probléma is van. Először is, olyan szavakat jeleníti meg, amik nem bírnak releváns jelentéssel, ugyanakkor gyakran fordulnak elő. Ezeket stop szavaknak hívjuk. A másik gond pedig az, hogy többször is megjeleníti ugyanazokat a szavakat, több szóalakban. Ilyen szó például az “olyan”. Itt legalább két formában tűnik fel: “ollyan” és “olyan”. Ezeket a problémákat megpróbálom megoldani egyrészt a megadott R függvények segítségével, másrészt szótár alkalmazásával, hogy még jobb eredményt érjünk el.

Szótár nélkül

A következő transzformációkat végeztem a szövegen, meglévő R scriptek segítségével:

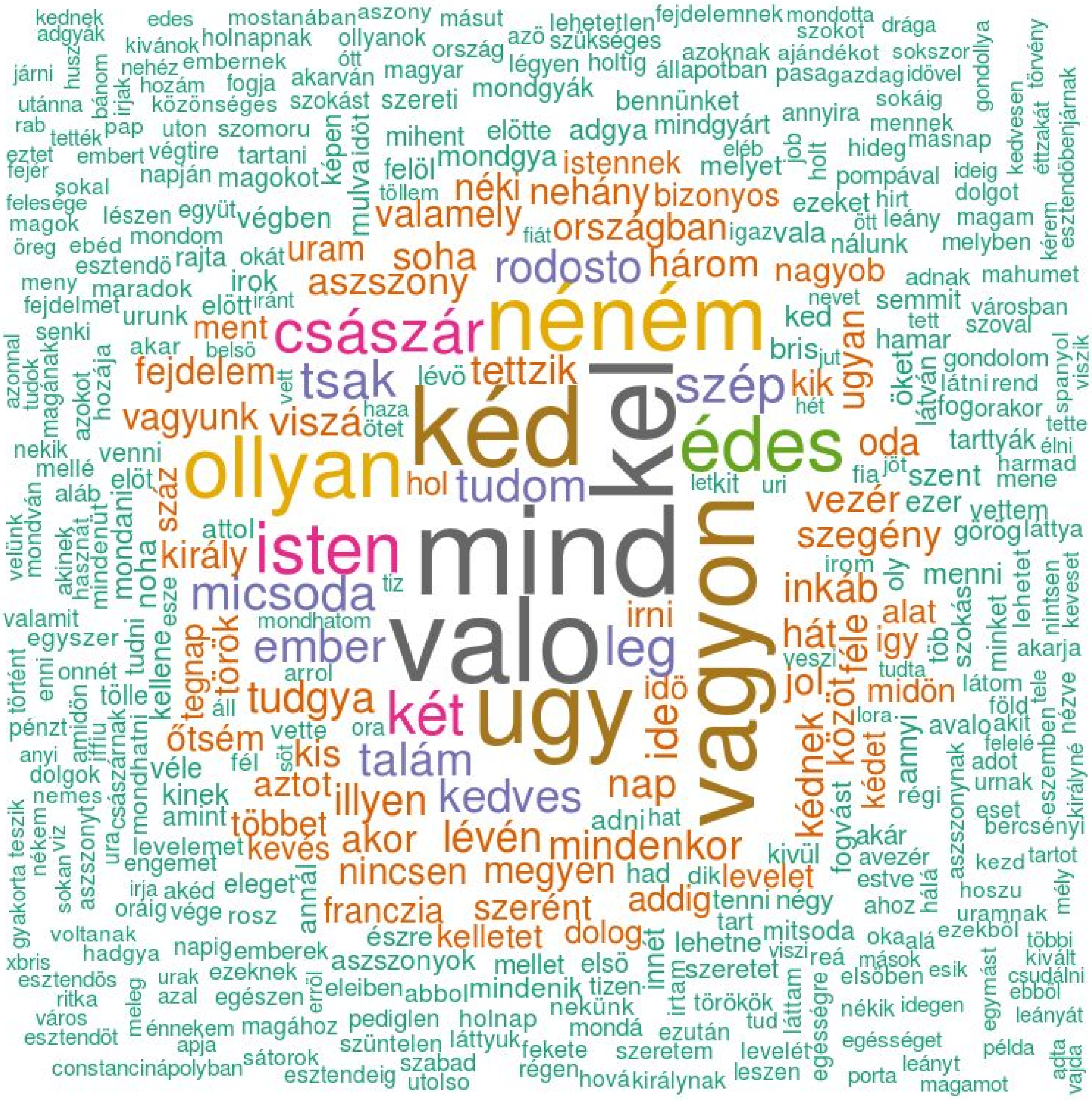
- Kitöröltem az írásjeleket
- Eltávolítottam a stop szavakat
- Kitöröltem a számokat
- Kis betűssé alakítottam a szöveget
- Eltűntettem a whitespace karaktereket

Ezeket a transzformációkat a demonstráció kedvéért egy végpontba összegyűjtöttem, ami vissza is adja a transzformált szöveget. Ezt láthatjuk a 13. ábrán.



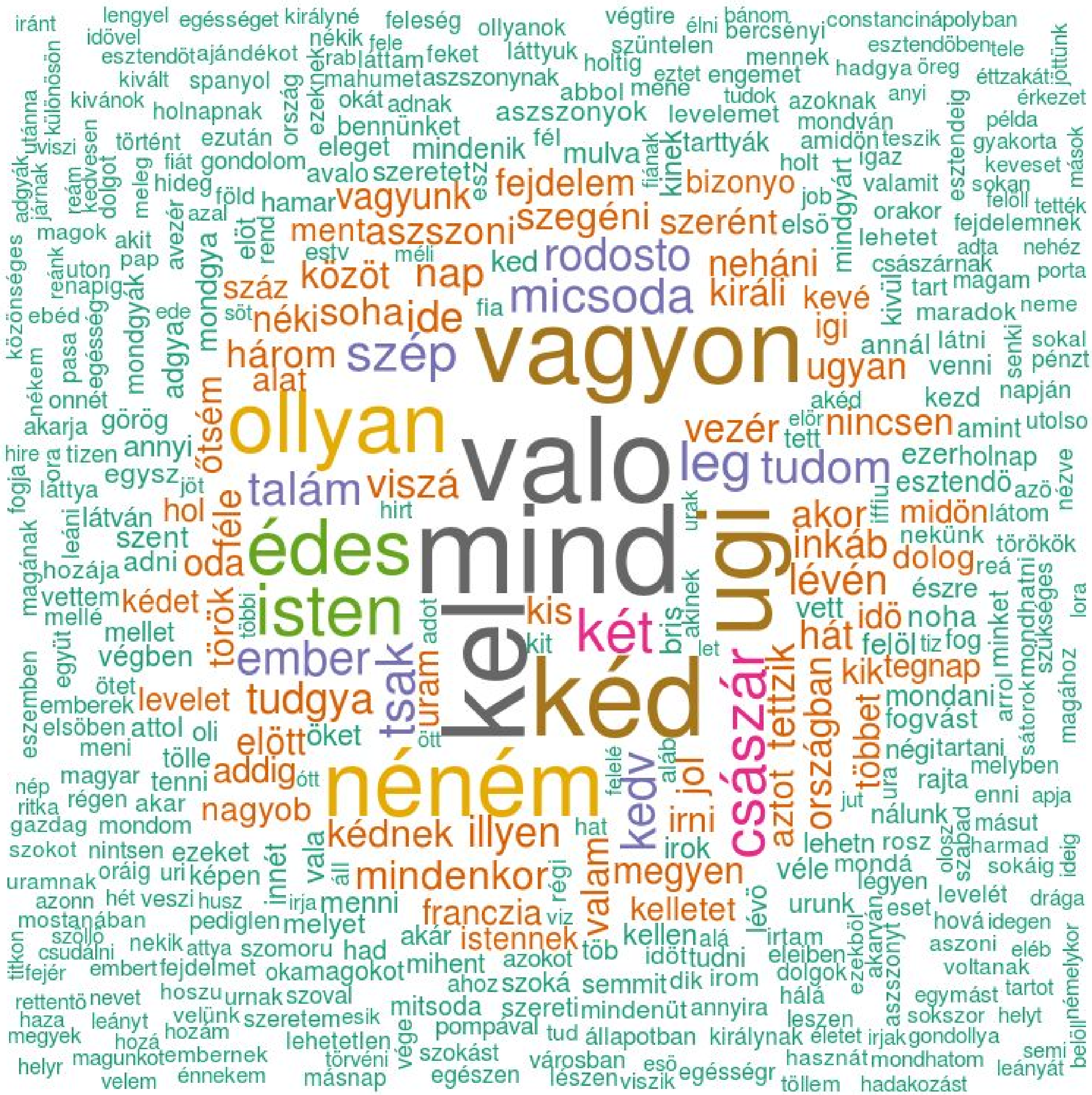
13. ábra. Transzformációk elvégzése utáni szövegrészlet

Ez után megint alkalmaztam a szó felhő generálását. 14. ábra mutatja az új eredményt:



14. ábra. Transzformált szövegre alkalmazott szófelhő eredménye

Itt már jobban látszik, hogy miért van szükségünk a szöveg normalizálására. Amennyiben előtte használjuk a beépített stemming algoritmust, az sem segít sokat. Ez látható a 15. ábrán.

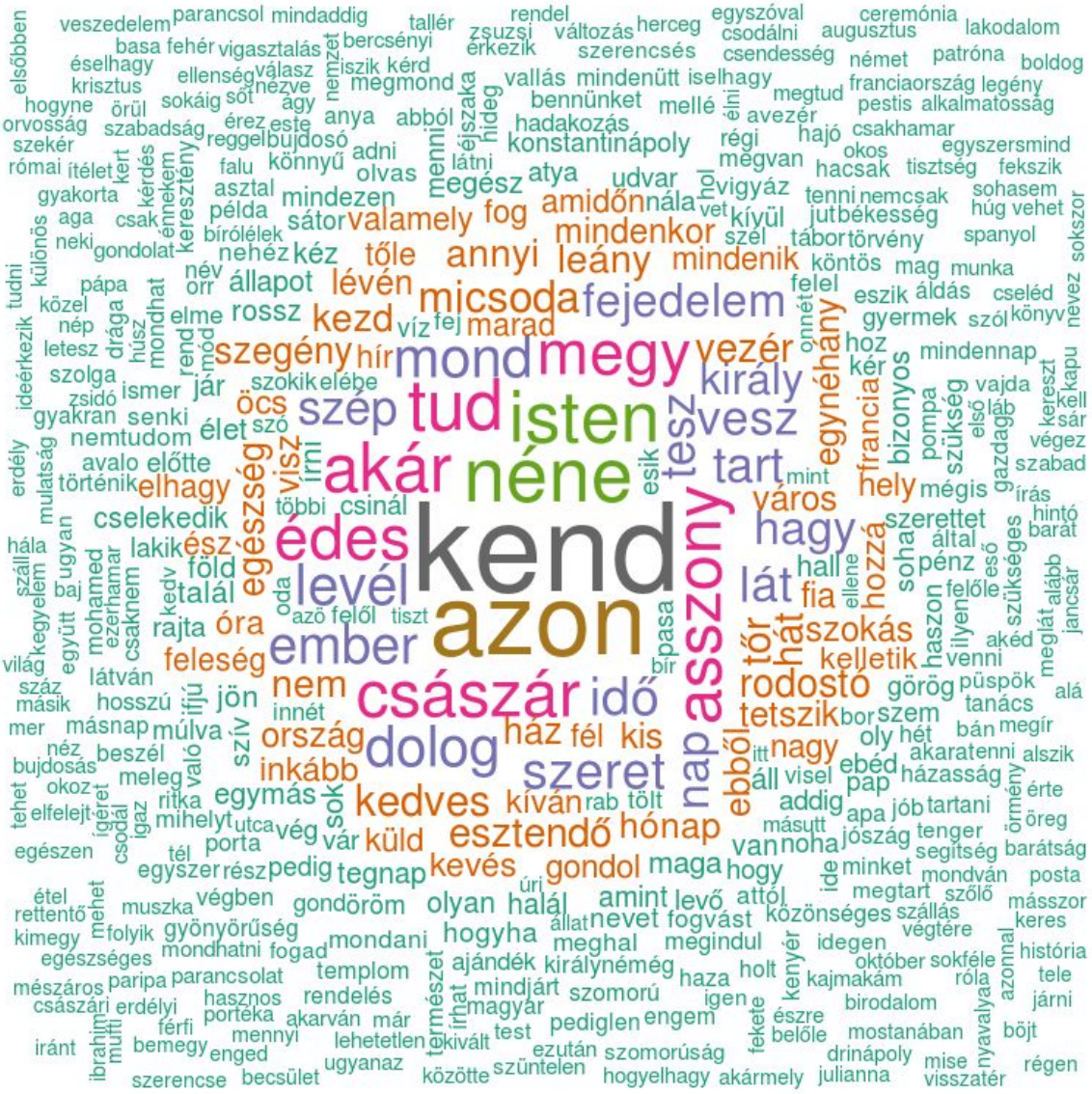


15. ábra. Beépített stemmer által normalizált szöveg szófelhője

Sajnos a beépített algoritmus tehetetlen a régies szavakkal szemben. Ilyenkor már kevésbé számít, hogy az milyen hatékony futási idő szempontjából. Nézzük meg, hogy mi történik a megfelelő szótár alkalmazásával.

Szótár alkalmazásával

Amennyiben először alkalmazzuk a stemming műveletet saját szótárunk segítségével, utána elvégezzük a szükséges transzformációkat, akkor sokkal jobb minőségű eredményt érünk el. 16. ábra mutatja be az újabb szó felhőt.



16. ábra. Szótár segítségével normalizált szöveg szófelhője

Itt már szerencsére a releváns szavak jelennek meg, amik jellemzik a művet.

További fejlesztési lehetőségek

Alábbiakban leírom a számomra legfontosabbnak tűnő fejlesztési lehetőségeket. Ezek ugyan nem szükségesek a feladat demonstrálásához, de az éles szoftver környezetben sokat számítanak.

Funkcionalitás

Egyrészt, bővíteni lehet a beépített szövegelemzési műveletek halmazát, illetve paraméterezhetővé tenni azokat. Másrészt, sokat jelenthet a szöveg normalizálási algoritmus hatékonyságának növelése.

Nyelvtan

A bővített funkcionalitás magával vonja a természetes nyelvi interfész kiterjesztését. Mivel azt szeretnénk, hogy minden műveletet el lehessen érni természetes nyelvi parancsokkal, így a kéréseket leíró nyelvtant folyamatosan bővíteni kell az újabb műveletek hozzáadásával.

A nyelvtant nem csak a műveletek lefedésére lehet fejleszteni, hanem a meglévő funkciók használatához rugalmasabb elérési lehetőséget biztosítani. Ez azt jelenti, hogy többféleképpen lehessen kifejezni ugyanazokat a kéréseket, és azokat felhasználó barát módon definiálni.

Webes front-end

Egyelőre csak JSON objektumok formájában adom vissza az eredményt. Teszteléshez és az elvek bemutatásához ez elegendő is, viszont a végső szoftverhez elengedhetetlen valamilyen felhasználó közeli front-end fejlesztése. A webes alkalmazás fejlesztéséhez egyaránt lehetőségünk van az előre definiált REST API végpontok használatához, amiket szokványos, grafikus eszközök segítségével érhetjük el, és a természetes nyelvi parancsokat kezelő interfészhez.

Ontológiák használata

Ahhoz, hogy a felhasználó számára több információval szolgáljunk, egyik fejlesztési lehetőség az ontológiák beintegrálása a rendszerbe. Például, a földrajzi helységeket és személyeket leíró tudásbázis felhasználása hasznosnak bizonyulhat az előbb bemutatott felhasználási területen. Ilyen módon, amennyiben valaki rákeresne egy helység nevére, akkor nem csak a nyers találatokat kapna, hanem azt is, hogy a tudásbázis alapján mely híres személyeknek függnek össze az adott helységgel.

Kitekintés

A szemantikus web egy aktív fejlesztési terület, amely 1990-s éves óta folyamatosan fejlődik. Egyre nagyobb igény jelenik meg a tudásbázisok beintegrálására. Erre láthattunk megoldást RDF/OWL formában leírt adatokban, amelyek szemantikus jelentést rendelnek a kifejezésekhez.

Az ilyen jellegű ontológiák tudással dúsíthatják a nyers szövegeket és hatékonyabbá tehetik a keresést és szövegfeldolgozási műveleteket. Láthattunk példát arra, hogy még egy egyszerű szótár használata is sokkal jobb eredményekhez vezethet.

Reményeim szerint sikerült mutatnom egy lehetséges megoldást a szövegelemző-rendszert felhasználó barátibb megközelítéshez, amivel megkönnyíthetjük az informatikában nem jártas felhasználók munkáját. Mivel számos bölcsésznek és természet tudósnak van szüksége ilyen eszközökre, ez egy fontos szempont.

A REST API interfész mellett, itt jelentős szerepet kap a természetes nyelvi interfész. A nyelvtan kiterjesztése nagy mértékben egyszerűbbé válik az elemző generátor alkalmazásával. Annak ellenére, hogy kevésbé hatékony, mint a szokásos LL(k) algoritmusok, az Earley algoritmus implementálása miatt bővebb nyelvtanra van lehetőségünk.

Irodalomjegyzék

1. Ontology-Based Interpretation of Keywords for Semantic Search
2. Semantic biomedical resource discovery: a Natural Language Processing framework
3. OWL
4. RDF
5. Open semantic search
6. Apache Stanbol
7. Evaluating four of the most popular Open Source and Free Data Mining Tools
8. Elasticsearch
9. Flask
10. Jinja2
11. REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces
12. Rserve
13. pyRserve
14. Chart Parsing
15. An Efficient Context-free Parsing Algorithm For Natural Languages
16. JavaCC