Integrációs és ellenőrzési technikák (VIMIAC04)

I. házi feladat: Szemantikus információkeresés

Készítette: Förhécz András, Strausz György
2020. április

Tartalom

1. Bevezetés

1.1. Az alkalmazási terület

A feladat célja bemutatni egy egyszerűsített feladat keretében a szemantikus modellezés előnyeit a szöveges dokumentumok keresése területén.
A házi feladatban egy kitalált alkalmazási terület dokumentumait kell indexelni, illetve keresni. Az alkalmazási terület a HWSW Informatikai Hírmagazin 2008-2009-es híreinek szövegbázisa. Az elkészítendő megoldás alapvető feladata, hogy képes legyen a rövid hírek szövegeiben egyszerű módon keresni.
A korpuszt (szöveggyűjteményt) egy tömörített állományként érhető el. Az állományon belül külön könyvtárakba vannak rendezve a cikkeket rovat (digitális otthon, ecotech, high-tech stb.), év és hónap szerinti bontásban. Az egyes cikkek szöveges állományok, melyek első sora a cikk címe, további sorai pedig a cikk törzse.

2. Követelmények

- Végezze el az alábbiakban leírt feladatokat. Használhat saját környezetet vagy virtuális gépet a fured.cloud.bme.hu felhőben található „Szemantikus web HF” nevű sablon

alapján. A virtuális gépen a feladat megoldásához szükséges környezetek és adatfájlok közvetlenül is elérhetőek.
- Készítsen jegyzőkönyvet a feladatoknál leírt követelmények szerint tetszőleges formátumban.
- A tárgyhoz tartozó IMSC pontokat az opcionális 4. feladat elvégzésével lehet megszerezni.
- Az elkészített jegyzőkönyv PDF nyomtatását és az elkészített projektet töltse fel egy ZIP fájlba tömörítve a hf.mit.bme.hu portálra a tárgy megfelelő feladatához.
- A feladatot Java, C++ vagy Phyton eszközök felhasználásával is megoldhatja, a segédprogramok Java Eclipse környezetben készültek.
- Határidő: 2020. április 26.

3. Feladatok

3.1. Egyszerű indexelés

A feladat: tervezzen és valósítson meg egy egyszerű inverz dokumentum előállító programot! A program a paraméterként megadott könyvtárban található szöveges dokumentumokat elemzi, majd egy kimeneti index fájlban eltárolja, hogy melyik szó mely dokumentumokban található meg.
Leadandó: az indexelő program és rövid leírása.
Felhasználható adatok: a korpusz.
Eszközök: A korpusz beolvasásában Java nyelven írt segédfüggvények segítenek, lásd a Segédleteket (5.4).
Tanácsok: Egy egyszerű statisztikát kell készíteni. Az inverz dokumentum (avagy dokumentum index) lényege az, hogy ha megadunk egy szót, akkor kikereshető legyen, hogy az mely dokumentumokban szerepelt. Azt is célszerű eltárolni, hogy egy szó hányszor szerepelt egy adott dokumentumban. Az index ne tartalmazza a gyakran előforduló nem tárgyterülethez tartozó szavakat (pl. névelők, kötőszavak), az ún. stop szavakat, ezeknek egy felhasználható listáját megtalálja a mellékletben.
Az index fájl szerkezete szabadon kialakítható, de feleljen meg a további feladatoknak. Az index kialakításakor ne mátrixos formában tárolja a szó-dokumentum párokat, hanem tömöritve, azaz csak azok a szó-dokumentum párok szerepeljenek, melyek előfordulási gyakorisága nem nulla.

3.2. Egyszerű keresés

A feladat: készítsen az 1. feladatban előállított index fájlra egy kereső programot. A program bemenő paraméterként szavakat kap, eredményként kiírja azon dokumentumok nevét, amelyekben mindegyik szó szerepel.
Leadandó: a kereső program, rövid magyarázat, futási példa.
Felhasználható adatok: az 1. feladatban készített index fájl.
Tanácsok: több szóból álló keresési minta esetén a dokumentumokat lehet szavanként is listázni, bár egy jó kereső rangsorolja az eredményt aszerint, hogy egy-egy dokumentumban hány szót talált meg (illetve melyik szó hányszor szerepelt). Ez a későbbi feladatok szempontjából is hasznos lehet.

3.3. Szemantikus keresőszó-kiegészítés

A találati pontosság javításának módszere a keresőszó-kiegészítés (*query expansion*, Wikipedia). Ilyenkor a felhasználó által megadott kulcsszavakhoz további, releváns kulcsszavakat adunk hozzá, és az ilyen módon kiegészített kulcsszó-lista alapján keresünk dokumentumokat.
A kiegészítő, releváns kulcsszavak kiválasztása történhet többféle módszerrel, itt egy szemantikus információn alapulót megközelítést választunk: a segédletek között megtalálható (5.4) PC-shop ontológia alapján egészítse ki a kulcsszavakat. Ha egy kulcsszó szerepel az ontológiában, akkor az ahhoz közeli fogalmak relevánsnak tekinthetőek.

A feladat: Módosítsa a 2. feladatban elkészített kereső programot szemantikus keresőszó kiegészítéssel! A felhasználó által megadott keresőszavakhoz vegyen fel további releváns kulcsszavakat (például specifikusabb fogalmak (osztályok leszármazottai), szinonimák (címkékben felsorolva)) a PC-shop ontológia és egy OWL következtető segítségével. A keresést érdemes úgy fontatni, hogy minden keresőszóhoz legalább egy releváns kulcsszó megtalálását megköveteli.
A keresést először futassa a segédletben található ontológián, majd az eredmények alapján egészítse ki az ontológiát, vegyen fel további elmeket a gyakori keresések eredményének javításához. Az ontológia módosításához használhatja a Protege szerkesztő eszközt.
Leadandó: a keresőszó kiegészítés módszerének rövid leírása, a módosított programok, a bővített ontológia, példa futtatások keresőszavakkal és a program kimenetével.
Felhasználható adatok: az előző feladatban elkészített programok, valamint indexállományok, a PC-shop ontológia.
Tanácsok: kiindulásként használja a Pellet OWL következtető példaprogramját (lásd segédlet).

4. IMSC feladat - súlyozott indexelés és keresés (opcionális)

Ezt a feladatot csak annak szükséges megoldani, aki a tárgyban IMSC pontokat kíván megszerezni.
A feladat: Módosítsa a korábbi feladatokban elkészített programokat úgy, hogy a szavak súlyát egy fejlettebb algoritmussal, a súlyozott indexeléssel állapítja meg. (A TF-IDF súlyozást lásd a segédletek között (5.3 pont)). Elemezze, hogy a súlyozott indexelés használata milyen előnyökkel jár az egyszerű indexeléshez képest.
Leadandó: A módosított programok, példa futtatások keresőszavakkal és a program kimenetével.
Tanácsok: A súlyok számítása a segédletben is ismertetett TF-IDF logaritmusos képlettel történjen. A keresésnél határozzon meg egy súlyokra épülő olyan relevancia mértéket, amivel a dokumentumok listája megfelel az elvárásainak (azaz egy keresési mintára valóban a legrelevánsabb dokumentumokat kapja meg elsőként).

5. Segédletek

5.1. Korpusz

A korpuszt az alábbi címen érik el:
HWSW 2008-2009 (5223 cikk): https://share.mit.bme.hu/index.php/s/Q9MCDspKwpBEE3R

A feladatok megoldása során a tesztelések céljára érdemes egy kisebb korpuszt létrehozni néhány cikk felhasználásával.

5.2. Segédprogram a korpusz beolvasásához

Az informaciokereses.zip projekten (https://share.mit.bme.hu/index.php/s/mo9A3EwENF7HpY6) belül a dokumentumok beolvasásához használható segédfüggvényeket találnak.

```
package hu.bme.mit.iir.Util;

public static List<String> readLinesIntoList(String fname) throws IOException
public static String readFileAsString(String fname) throws IOException
```

A Util osztályban a fenti két függvény szöveges fájl beolvasására használható. Egyetlen paraméterük a fájl elérési útja. Az első esetben az egyes sorok egy listába kerülnek, míg a második függvény egyetlen String-ként adja vissza a fájl tartalmát.

```
package hu.bme.mit.iir.TermRecognizer;

TermRecognizer() throws IOException
TermRecognizer(Set<String> stopwords)
TermRecognizer(Set<String> stopwords, String delimiters)
Map<String, Integer> termFrequency(String text)
```

A TermRecognizer a cikkek termekre (kulcsszavakra) bontásában segít. Az objektum létrehozható a figyelmen kívül hagyott szavak (*stopword*) listájának megadásával és a szavakat elválasztó karakterek (*delimiters*) megadásával. Ha nem adjuk meg ezeket, a lokális könyvtárban található "stopwords.txt"-t próbálja meg beolvasni.

A **termFrequency** függvénnyel egy szöveget termekre bonthatunk. Az eredményül kapott java.util.Map struktúrában az egyes termekhez a gyakoriságuk lesz hozzárendelve. Például az alábbi szövegre:

A Béla és a Peti meg a Béla elmentek fagyizni.
A visszaadott eredmény (feltéve, hogy az "A", "és", "a" valamint "meg" figyelmen kívül hagyott szavak):

Béla -> 2
elmentek -> 1
fagyizni -> 1
Peti -> 1

5.3. OWL következtető használata

Az OWL következtető használtalához töltse le a HermiT következtetőt:
https://share.mit.bme.hu/index.php/s/dzDDwQRzXHS9Qcd
(A példa Eclipse projekt tartalmazza.)

A letöltött állomány kicsomagolása után a HermiT.jar-t fel kell venni a Java classpath-ba.

A következtetőt az OWL API-n (http://owlapi.sourceforge.net/)keresztül érjük el. Az OWL API dokumentáció releváns részei az alábbiak:

- OWL API reference (Javadoc) http://owlapi.sourceforge.net/documentation.html
- remek (angol nyelvű) mintapéldák és tutorial az OWL API dokumentáció honlapján (a feladathoz nem feltétlenül szükséges)

Valójában bármely más OWL API-kompatibilis következtetőt (pl. Pellet, Fact++) is használhatnánk azonos programkóddal, csak a következtető inicializálását végző sort kellene kicserélni. A HermiT következtetőt az alábbi sorral lehet inicializálni:

```
OWLReasonerFactory reasonerFactory = new org.semanticweb.HermiT.Reasoner.ReasonerFactory();
```

Az alábbiakban közlünk egy kiindulásként felhasználható mintapéldát az OWL API és a HermiT használatához. A példaprogram beolvassa a PC-Shop ontológiát, majd az osztályhierarchia alapján keres releváns kulcsszavakat az "alkatrész" termhez. További magyarázatot találnak a kommentekben.

A PelletReasoningExample példaprogram megtalálható az informaciokereses.zip projektben.

```java
001 package hu.bme.mit.iir;
002
003 import java.io.File;
004 import java.util.Collections;
005 import java.util.HashSet;
006 import java.util.Set;
007
008 import org.semanticweb.owlapi.apibinding.OWLManager;
009 import org.semanticweb.owlapi.model.*;
010 import org.semanticweb.owlapi.reasoner.*;
011 import org.semanticweb.owlapi.vocab.OWLRDFVocabulary;
012
013 import com.clarkparsia.pellet.owlapiv3.PelletReasonerFactory;
014
015 public class PelletReasoningExample {
016
017   public static final String PCSHOP_ONTOLOGY_FNAME = "pc_shop.owl.xml";
018   public static final String PCSHOP_BASE_URI =
019     "http://mit.bme.hu/ontologia/.....";
020   public static final IRI ANNOTATION_TYPE_IRI =
021       OWLRDFVocabulary.RDFS_LABEL.getIRI();
022
023   OWLOntologyManager manager;
024   OWLOntology ontology;
025   OWLReasoner reasoner;
026   OWLDataFactory factory;
027
028   public PelletReasoningExample(String ontologyFilename) {
029     // Hozzunk létre egy OntologyManager példányt. Többek között ez tartja
030     // nyilván, hogy mely ontológiákat nyitottuk meg és azok hogyan
031     // hivatkoznak egymásra.
032     manager = OWLManager.createOWLOntologyManager();
033
034     // Töltsük be az ontológiát egy fizikai címről.
035     // [Az ontológiáknak külön van logikai és fizikai címe, a kettő közötti
036     // leképezést az URIMapper-ek határozzák meg: manager.addURIMapper(...).
037     // Erre ontológiák közötti függőségek (import) kialakításánál van
038     // szükség.]
039     ontology = null;
040     try {
041       ontology = manager.loadOntologyFromOntologyDocument(
042           new File(ontologyFilename));
043     } catch (Exception e) {
044       System.err.println("Hiba az ontológia betöltése közben:\n\t"
045           + e.getMessage());
046       System.exit(-1);
047     }
048     System.out.println("Ontológia betöltve: " +
049         manager.getOntologyDocumentIRI(ontology));
050
051     // Létrehozzuk a következtetőgép egy példányát. Mi most a Pellet-et
052     // használjuk, de az OWL API univerzális, másik következtető
053     // használatához csak a Factory osztály nevét kellene kicserélni.
054     //
055     // A classpath-ban szerepelnie kell a Pellet jar file-jainak:
056     //   pellet-core.jar pellet-datatypes.jar pellet-explanation.jar, ...
057     OWLReasonerFactory reasonerFactory = new PelletReasonerFactory();
```

```
058     //
059     // Hozzunk létre egy következtetőgépet, betöltve az ontológiát.
060     // [Ha vannak az ontológiának függőségei, azok automatikusan
061     // betöltődnek a manager segítségével.]
062     reasoner = reasonerFactory.createReasoner(ontology);
063
064     try {
065       // Ha az ontológia nem konzisztens, lépjünk ki.
066       if (!reasoner.isConsistent()) {
067         System.err.println("Az ontológia nem konzisztens!");
068
069         Node<OWLClass> incClss = reasoner.getUnsatisfiableClasses();
070         System.err.println("A következő osztályok nem konzisztensek: "
071             + Util.join(incClss.getEntities(), ", ") + ".");
072         System.exit(-1);
073       }
074     } catch (OWLReasonerRuntimeException e) {
075       System.err.println("Hiba a következtetőben: " + e.getMessage());
076       System.exit(-1);
077     }
078
079     // Példányosítsuk az OWLDataFactory-t, aminek a segítségével létre tudunk
080     // hozni URI alapján OWL entitásokat (osztályt, tulajdonságot, stb).
081     factory = manager.getOWLDataFactory();
082   }
083
084   // Lekérdezi egy osztály leszármazottait a PC-Shop ontológiában.
085   //   className: a keresett osztály neve
086   // Az OWL következtető Set<Set<..>> struktúrákban, azaz halmazok
087   // halmazaként adja vissza a lekérdezés eredményét. A belső halmazok
088   // egymással ekvivalens, jelentés szempontjából megegyező dolgokat
089   // tartalmaznak. Például egy lekérdezés eredménye:
090   // { { Kutya, Eb }, { Macska, Cica }, { Ló } }
091   // Ez a függvény a fenti halmazt "kilapítva" adja vissza:
092   // { Kutya, Eb, Macska, Cica, Ló }
093   public Set<OWLClass> getSubClasses(String className, boolean direct) {
094     // Létrehozzuk az osztály URI-ját a base URI alapján.
095     IRI clsIRI = IRI.create(PCSHOP_BASE_URI + className);
096     // Ellenőrizzük, hogy az osztály szerepel-e az ontológiában.
097     // (Ha olyan dologra kérdezünk rá a következtetővel, ami nem szerepel az
098     // ontológiában, az nem vezet hibához az OWL nyílt világ feltételezése
099     // miatt.)
100     if (!ontology.containsClassInSignature(clsIRI)) {
101       System.out.println("Nincs ilyen osztály az ontológiában: \"" +
102           className + "\"");
103       return Collections.emptySet();
104     }
105     // Létrehozzuk az osztály egy példányát és lekérdezzük a leszármazottait.
106     OWLClass cls = factory.getOWLClass(clsIRI);
107     NodeSet<OWLClass> subClss;
108     try {
109       subClss = reasoner.getSubClasses(cls, direct);
110     } catch (OWLReasonerRuntimeException e) {
111       System.err.println("Hiba az alosztályok következtetése közben: "
112           + e.getMessage());
113       return Collections.emptySet();
114     }
```

```
115     // Kiírjuk az eredményt, az ekvivalens osztályokat
116     // egyenlőségjellel elválasztva.
117     System.out.println("Az \"" + className + "\" osztály leszármazottai:");
118     for (Node<OWLClass> subCls : subClss.getNodes()) {
119       System.out.println("  - " + Util.join(subCls.getEntities(), " = "));
120     }
121     return subClss.getFlattened();
122   }
123
124   // Lekérdezi egy OWL entitás (osztály, tulajdonság vagy egyed) annotációit
125   // az ontológiából. (Ehhez nincs szükség a következtetőre.)
126   // Az annotációknak sok fajtája van, a lekérdezett típust az
127   // ANNOTATION_TYPE_IRI konstans tartalmazza, jelen esetben rdfs:label.
128   public Set<String> getClassAnnotations(OWLEntity entity) {
129     OWLAnnotationProperty label =
130         factory.getOWLAnnotationProperty(ANNOTATION_TYPE_IRI);
131     Set<String> result = new HashSet<String>();
132     for (OWLAnnotation a : entity.getAnnotations(ontology, label)) {
133       if (a.getValue() instanceof OWLLiteral) {
134         OWLLiteral value = (OWLLiteral)a.getValue();
135         result.add(value.getLiteral());
136       }
137     }
138     return Collections.unmodifiableSet(result);
139   }
140
141   // Példaprogram: beolvassa a pc-shop OWL ontológiát, majd listázza
142   // az "alkatrész" osztály valamennyi leszármazottját és azok annotációit.
143   public static void main(String[] args) {
144     // Példányosítsuk a fenti egyszerű Pellet következtető osztályt.
145     PelletReasoningExample p = new PelletReasoningExample(
146         PCSHOP_ONTOLOGY_FNAME);
147
148     // Végezzük keresőszó-kiegészítést az "alkatrész" kulcsszóra
149     // az osztály leszármazottai szerint!
150     final String term = "alkatrész";
151     Set<OWLClass> descendants = p.getSubClasses(term, false);
152     System.out.println("Query expansion a leszármazottak szerint: ");
153     for (OWLClass cls : descendants) {
154       // Az eredmények közül a beépített OWL entitásokat ki kell szűrnünk.
155       // Ezek itt az osztályhierarchia tetejét és alját jelölő
156       // "owl:Thing" és "owl:Nothing" lehetnek.
157       if (!cls.isBuiltIn()) {
158         // Kérdezzük le az osztály címkéit (annotation rdfs:label).
159         Set<String> labels = p.getClassAnnotations(cls);
160         System.out.println("\t- "
161             + term + " -> " + cls.getIRI().getFragment()
162             + " [" + Util.join(labels, ", ") + "]");
163       }
164     }
165   }
166 }
```

5.4. TF-IDF index

A szöveg objektumokból álló mintát egy kulcsszó-dokumentum mátrixszal lehet ábrázolni, ahol a sorok a kulcsszavaknak, az oszlopok az objektumoknak felelnek meg. A mátrix bejegyzések az adott

kulcsszó adott objektumbeli előfordulásának felelnek meg. Amennyiben a kulcsszó nem fordul elő az adott objektumban a bejegyzés nulla, előfordulás esetén nemzérus, a legegyszerűbb esetben egy (bináris mátrix).
A nyelvi kifejezés jobb közelítése érdekében lokális és globális súlyokat vezethetünk be. Lokális súly az aktuális dokumentumra vonatkozik, tipikusan a kulcsszó dokumentumbeli előfordulási száma. Globális súly a kulcsszóra vonatkozik, a kulcsszó specifikusságát adja meg.
Egy szokásos képlet a *w* kulcsszó *i* dokumentumon belüli súlyának megadására:

$$weight(w,i) = \frac{f_{wi}\log(N/n_w)}{\sqrt{\sum_{k=1}^{W_i}(f_{ki})^2(\log(N/n_k))^2}}$$

ahol f_{wi} *w* előfordulási száma *i*-ben, *N* az összes dokumentum száma, n_w azon dokumentumok száma, amikben szerepel *w*, W_i az *i*-ben szereplő kulcsszavak száma. Azaz a súly egyenesen arányos a kulcsszó előfordulási számával az adott dokumentumban. A számlálóban levő másik tag akkor ad nagy értéket, ha minél kevesebb dokumentumban szerepel a kulcsszó (ha minden dokumentumban szerepel, akkor a kapott súly nulla). Mivel a vektor normája függ a benne levő kulcsszavak számától és súlyától, ezért a nevezőben levő taggal normalizáljuk.
A lekérdezés során a lekérdezési és dokumentum vektorokat kell összehasonlítani. Amennyiben a két dokumentum hasonló, akkor valószínűleg sok közös kulcsszóval rendelkeznek, ezért a két vektor közel helyezkedik el egymáshoz. Ezt a közelséget általában a két vektor által bezárt szög koszinuszával fejezik ki:
C = *a* * *b* / (|| *a* || * || *b* ||)
ahol *a* és *b* a dokumentumok vektorai, * jelöli a skaláris szorzást és || || az euklideszi normát.
Ha a kulcsszavak száma *n*, az még nem jelenti azt, hogy minden egyes dokumentumot *n* súly ír le, mert a legtöbb súly nulla. Ennek következtében nincs sokkal több memóriára szükség, mint a Bool modell esetén.
A modell előnyei:

- a lekérdező vektorban lehet súlyokat rendelni a kulcsszavakhoz,
- a hasonlósági mérték alapján lehet relevancia-sorrendbe rendezni a visszaadott dokumentumokat.

A modell hátrányai:

- a kulcsszavaknak függetlennek kell lenniük,
- nem lehet logikai kapcsolatokat felírni a kulcsszavak között.

A módszer használható számos alkalmazási területen. Például kórlapok illetve ez alapján egyes orvosi esetek hasonlóságának vizsgálatára vagy egyes kódolást támogató rendszerek is hasonló elven működnek, megkeresik a kódolandó szöveghez legjobban hasonlító kódolt dokumentumot, és veszik annak kódját.

5.5. Ajánlott szoftverek, adatfájlok listája

- Mintapélda és segédprogramok egy Eclipse projektként: https://share.mit.bme.hu/index.php/s/mo9A3EwENF7HpY6
- Hermit következtető: https://share.mit.bme.hu/index.php/s/dzDDwQRzXHS9Qcd
- PC-shop ontológia: https://share.mit.bme.hu/index.php/s/Q9APXWsGwBipeJN
- Stop szavak listája: https://share.mit.bme.hu/index.php/s/b7ePSzC3ktppnad
- Korpusz: https://share.mit.bme.hu/index.php/s/Q9MCDspKwpBEE3R