Relazione TLN Terza Parte

Maltese Walter, Morelli Pierandrea

Esercizio 1.1

- Obiettivo: calcolo della similarità tra l'insieme di definizioni associate ai 4 concetti dati, di cui 2 concreti (uno generico e uno specifico) e altri 2 astratti (uno generico e uno specifico).

- Implementazione:

  1. Pre-processing delle definizioni -> rimozione stopwords, punteggiatura, lemmatizzazione.

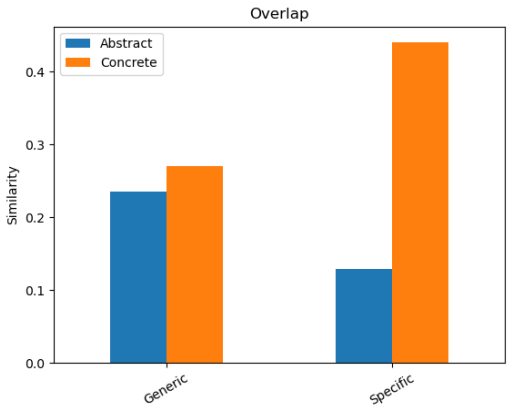
```
# Removing stopwords
definition = definition.lower()
stop_words = set(stopwords.words('english'))
punct = {',', ';', '(', ')', '{', '}', ':', '?', '!', '.'}
wnl = nltk.WordNetLemmatizer()
tokens = nltk.word_tokenize(definition)
tokens = list(
    filter(lambda x: x not in stop_words and x not in punct, tokens)) #returning only the "clean" tokens

# Lemmatization
lemmatized_tokens = set(wnl.lemmatize(t) for t in tokens) #lemmatization of the "clean" tokens returned

return lemmatized_tokens #set of lemmas (cleaned after preprocessing)
```

     Abbiamo utilizzato a tal fine l'approccio a "bag of words", visto già nella seconda parte del corso.

  2. Calcolo similarità -> computazione dell'overlap tra due definizioni, ripetendo questo processo per tutte le definizioni di un dato termine. Infine effettuiamo una normalizzazione per avere un risultato nell'intervallo [0,1].
     Come prima, questo approccio tramite overlap è stato ripreso dagli svolgimenti della seconda parte del corso.

```
i = 0
while i < len(definitions):
    a = bag_of_words(definitions[i])  # set of terms (after preprocessing) of the first definition
    j = i + 1
    while j < len(definitions) - 1:
        b = bag_of_words(definitions[j])  # set of terms of the second definition
        # Computing similarity between definitions
        t = len(a & b) / min(len(a), len(b)) # overlap result with normalization [0,1]
        results.append(t) #inserting overlap results in list result
        j = j + 1
```

- Risultati: notiamo che i concetti concreti hanno generalmente uno score di similarità più alto rispetto a quelli astratti, ma tale differenza risulta molto più marcata quando si confrontano concetti specifici (concreti-specifici vs astratti-specifici), rispetto al risultato ottenibile dal confronto di (score di similarità relativi a) oggetti generici (concreti-generici vs astratti-generici).




```
Esercizio 1.1 - Similarità:

          Abstract Concrete
Generic       24%      27%
Specific      13%      44%
```

Esercizio 1.2

- Obiettivo: spiegare i valori di similarità ottenuti nello step precedente.
- Implementazione: usufruendo di “Counter()”, abbiamo ricavato le 3 parole più frequenti nelle varie definizioni, e associata a ciascuno di essi una frequenza, corrispondenza alla percentuale di occorrenze di quel termine nelle definizioni.

```
Esercizio 1.2 - Similarity Explanation:

most frequent words for generic_abstract = [('ability', '60.0%'), ('fear', '53.33%'), ('face', '30.0%')]
most frequent words for generic_concrete = [('material', '70.0%'), ('used', '43.33%'), ('writing', '26.67%')]
most frequent words for specific_abstract = [('fear', '33.33%'), ('anxiety', '33.33%'), ('something', '30.0%')]
most frequent words for specific_concrete = [('pencil', '83.33%'), ('sharpen', '53.33%'), ('tool', '53.33%')]
```

Esercizio 1.3

- Obiettivo: integrazione tra PropertyNorms e WordNet (scelta la seconda opzione dell’esercizio).
  - Le risorse in PropertyNorms descrivono delle proprietà percettive/cognitive, ossia che vengono in mente alla descrizione di un dato concetto e derivano da questionari a cui veniva chiesto alle persone di descrivere proprietà/caratteristiche di alcune parole che trovavano scritte.
    Sono feature semantiche basate su percezione e immediatezza. Ad esempio, pensando a canarino le prime parole che ci vengono in mente sono: giallo, canta ecc.
  - Wordnet, d’altro canto, rappresenta un database semantico-lessicale per la lingua inglese, che ha lo scopo di organizzare, definire e descrivere i concetti espressi dai vocaboli, raggruppandoli in termini con significato affine, chiamati “synset”, e collegando i vari significati attraverso diversi tipi di relazioni chiaramente definite.
- Implementazione:
  1. Creazione dizionario contenente (per ogni concetto nella risorsa) tutte le feature presenti nella risorsa di PropertyNorms (dal file “property_norms_cut”).
  2. Creazione set contenente le informazioni reperite da WordNet riguardo i “supersensi" (categorie lessicali di riferimento) e relative definizioni, per tutti concetti considerati.
  3. Processing delle informazioni reperite da entrambe le risorse.
  4. Intersezione tra i due set (tra le feature di PropertyNorms e le definizioni reperite da WordNet)

```python
def wn_definitions(word):
    word_senses = wn.synsets(word) #taking wordnet synsets of the given parameter "word"
    wn_definition = set() #list for storing lexical category and relative definitions
    # Synset('noun.animal.01') (= word_senses[0]) -> noun.animal (= lexname) -> animal (= .split('.')[1])
    lexical_category = word_senses[0].lexname().split('.')[1] #from word=dog to hypernomim=animal
    wn_definition.add(lexical_category) #adding hyperonim (lexical category of the term "word" in WordNet)
    #print (wn_definition)
    for sense in word_senses:
        wn_definition.update(aux(sense.definition())) #adding preprocessed hyperonims glosses (clean zebratokens of the definitions
    #print (wn_definition)
    return wn_definition

def intersection(word):
    "Return: intersection beetween word's features in PropertyNorms and WN definitions "
    return (words.get(word) & wn_definitions(word))
```

- Risultati: sulla shell stampiamo le feature presenti in entrambe le risorse.
  Queste, in particolare, saranno, fra tutte le feature associate al termine inserito in input, quelle che, oltre ad esser presenti su PropertyNorms, esistono anche su WordNet.
  Sarà infatti la terza colonna del file di output “result.csv” a indicarne quelle che rientrano in tale intersezione.

```
Please enter a word: zebra
{'animal', 'striped'}
```

```
Concept,Feature,Present_in_WN
zebra,horse,no
zebra,legs,no
zebra,tail,no
zebra,lions,no
zebra,fast,no
zebra,white,no
zebra,zoos,no
zebra,mammal,no
zebra,found,no
zebra,shy_timid,no
zebra,herbivorous,no
zebra,wild,no
zebra,ears,no
zebra,pretty_attractive_beautiful,no
zebra,herds,no
zebra,hunted_is,no
zebra,four,no
zebra,move,no
zebra,like,no
zebra,prey,no
zebra,striped,yes
zebra,grass,no
zebra,stripes,no
```

Esercizio 1.4

- Obiettivo: scegliere un verbo transitivo e individuare i cluster semantici più rappresentativi dei suoi due argomenti (soggetto e oggetto).
  Alla base dello svolgimento di questo esercizio vi è la cosiddetta "Teoria di Hanks”, che indica il verbo come radice del significato.
  Ad ogni verbo viene associata una valenza che indica il numero di argomenti necessari per il verbo. In base al numero di argomenti che un verbo richiede, in alcuni casi possiamo differenziarne il significato.
  Una volta determinato il numero di argomenti di un verbo dobbiamo specificarli mediante un certo numero di slot. Ogni slot può avere un certo numero di valori che lo riempiono, detti filler. Ogni filler può avere associati dei tipi semantici che rappresentano delle generalizzazioni concettuali strutturate come una gerarchia (zebra → animale → essere vivente...).
  Raggruppiamo i vari filler secondo alcuni types. Essi sono gruppi semantici generici o categorizzazioni/clusterizzazioni dei filler.

- Implementazione:

  1. Nella nostra implementazione abbiamo concesso la scelta tra due verbi transitivi, “to pay” e “to promise”. L’utente è invita a sceglierne uno per proseguire con l’analisi degli argomenti di tale verbo.

  2. Abbiamo estratto da un corpus (EnglishWeb2020), usufruendo di SketchEngine, i contesti (tutte le frasi) in cui il verbo scelto è utilizzato.

  3. Parsing: mediante il metodo “load(“en_core_web_sm")" di spacy, identifichiamo gli slot relativi al soggetto e al complemento oggetto delle frasi relative al verbo richiesto.

```
p_subj = {'subj', 'nsubjpass', 'nsubj'}
p_obj = {'pobj', 'dobj', 'obj', 'iobj'}

#parse sentences with spacy to find the subject and the object argument of the selected verb
def parse_find_subj_obj(sent, i):
    o = None
    s = None

    sent = nlp(sent) #using Language object "en_core_web_sm" with spacy on our sentences

    for elem in sent: #for all the elements of the sentence taken as parameter
            if elem.dep_ in p_subj: #dep_ = "Syntactic dependency relation" -> to find p_subj
                if elem.lemma_ != "-PRON-":
                    s = elem.lemma_ #subject found
                else:
                    s = elem.text
            if elem.dep_ in p_obj: #dep_ = "Syntactic dependency relation" -> to find p_obj (
                if elem.lemma_ != "-PRON-":
                    o = elem.lemma_ #object found
                else:
                    o = elem.text
    parsed_sent(sent)
    return s,o
```

4. Disambiguazione: disambiguiamo il senso corretto dei 2 argomenti del verbo (soggetto e oggetto) -> per il primo, se esso rientra fra i pronomi personali non abbiamo bisogno di utilizzare l'algoritmo di Lesk e restituiamo il primo (il più comune) synset di WordNet associato al termine People; altrimenti, utilizziamo l'algoritmo di Lesk per trovare il miglior senso possibile (il synset corretto) per il soggetto nel contesto dato.
   Riguardo l'oggetto, invece, proseguiamo con la disambiguazione utilizzando direttamente l'algoritmo di Lesk.

```
def wsd(sent, subj, obj):
    possible_subj = ["i", 'you', 'he', "she", "it", "we", "they"] #soggetti più comuni (pronomi personali) -> riconducibili al sup
    if subj in possible_subj: #se il soggetto in questione è di tipo "comune"
        ris = wn.synsets('people')[0]     #Prende il primo synset di WordNet associato al termine 'people'
    elif subj is not None: #se il soggetto in questione non è di tipo "comune" ma è != None -> necessaria analisi ulteriore
    #disambiguazione con algoritmo di lesk (soggetto) -> trova il miglior senso possibile per la parola "subj" nel contesto "sent"
        ris = lesk(sent, subj)
    else:
        ris = None
    if obj is not None:
    #disambiguazione con algoritmo di lesk (oggetto) -> trova il miglior senso possibile per la parola "obj" nel contesto "sent"
        ris1 = lesk(sent, obj)
    else:
        ris1 = None
    return ris, ris1 #ritorno la tupla (soggetto, oggetto) disambiguata
```

5. Recupero dei super-sensi (categorie lessicali) di soggetto e oggetto trovati (dopo il parsing e la disambiguazione delle frasi contenenti il verbo richiesto).
   In tal modo identifichiamo per i 2 argomenti del verbo dato, quali sono i semantic type che fanno da fillers per tali slot del verbo.

```
def super_sense(ris, ris1):

    if ris is not None or ris1 is not None:
        if ris is not None:
            ss1 = ris.lexname() #lexical category (super-sense) of the subject found
        else: #ris1 is None
            ss1 = None
        if ris1 is not None:
            ss2 = ris1.lexname() #lexical category (super-sense) of the object found
        else: #ris2 is None
            ss2 = None
    else:
        ss1 = None
        ss2 = None

    return ss1, ss2 #return a tuple of possible values (or None) for the super-senses of subject and object
```

- Risultati: stampiamo in output i 10 Semantic-Types (i supersensi) più frequenti, prima per le coppie dei 2 argomenti (soggetto,oggetto) e, poi, considerandoli singolarmente.

```
1 - To Pay;
2 - To Promise;
1
100%|██████████| 9677/9677 [00:40<00:00, 237.00it/s]
Semantic Type:   ('noun.group', 'noun.act') 5.34  %
Semantic Type:   ('noun.group', 'noun.cognition') 4.55  %
Semantic Type:   ('noun.group', 'noun.communication') 4.21  %
Semantic Type:   ('noun.group', 'noun.artifact') 3.86  %
Semantic Type:   ('noun.group', 'noun.possession') 3.56  %
Semantic Type:   ('noun.group', 'noun.person') 3.0  %
Semantic Type:   ('noun.group', 'noun.group') 2.11  %
Semantic Type:   ('noun.group', 'verb.possession') 2.02  %
Semantic Type:   ('noun.group', 'adj.all') 1.91  %
Semantic Type:   ('noun.group', 'noun.time') 1.77  %
Filler slot 1:   group 49.42  %
Filler slot 1:   person 10.47  %
Filler slot 1:   all 5.62  %
Filler slot 1:   substance 5.29  %
Filler slot 1:   communication 4.3  %
Filler slot 1:   act 3.61  %
Filler slot 1:   cognition 3.34  %
Filler slot 1:   possession 3.23  %
Filler slot 1:   artifact 3.07  %
Filler slot 1:   quantity 1.63  %
Totale slot 1  5538
Filler slot 2:   communication 11.01  %
Filler slot 2:   act 10.38  %
Filler slot 2:   cognition 10.24  %
Filler slot 2:   possession 10.22  %
Filler slot 2:   artifact 7.95  %
Filler slot 2:   person 6.66  %
Filler slot 2:   all 5.92  %
Filler slot 2:   group 3.9  %
Filler slot 2:   time 3.9  %
Filler slot 2:   attribute 3.59  %
Totale slot 2  5538
```

Esercizio 1.5

- Obiettivo: partendo dalle definizioni usate nell'esercizio 1.1/1.2 per descrivere i 4 concetti di Courage, Paper, Apprehension, Sharpener, vogliamo trovare un metodo automatico per risalire al vero concetto (al giusto synset), quindi siamo nel caso di un problema cosiddetto "content to form".

  Tali problemi identificano l' "Onomasiologic search", ovvero il problema di passare da qualcosa che abbiamo in mente concettualmente, a cui però non sappiamo dare un nome o non ce lo ricordiamo, alla definizione/forma del concetto inteso. Proprio da tali situazioni si giunge al ben noto "tip of the tongue problem".

- Implementazione:

  1. Processiamo le definizioni dei 4 concetti oggetto di studio, ottenendo the parole più comuni utilizzate al loro interno

  2. Passo del content-to-form: dalle definizioni determiniamo il WordNet synset corretto, quello inteso per il concetto:

```
        for genus in common_words: #for all 3 of the most commond words of a definition
            #best_sense = lesk(context, genus)
            #hypo_list = best_sense.hyponyms()
            hypo_list = getAllHyponyms(genus)
            overlaps_list.extend(getSynsetsOverlap(hypo_list, context)) #computing a list of tuples (synset, overlap)...
        overlaps_list.sort(key=lambda tup: tup[1], reverse=True) #...and sorting it (by the value of the overlap) by descending order
        #taking only the first elem (the WN synset of a certain hyponim of the genus) of the tuples (synset, overlap) in overlaps_list
        overlaps_list = [a_tuple[0] for a_tuple in overlaps_list]

        #returning only the first 10 synsets (the 10 hyponyms with the most similar context with the global context for the def considered)
        return overlaps_list[:10]
```

     1. Recupero i potenziali genus (iperonimi) per le definizioni processate
     2. Computo l'overlap tra il contesto degli iponimi di ognuno di questi potenziali genus identificati ed il contesto generale, comprendente l'integrità delle definizioni associate ad un certo concetto.
     3. Identifichiamo come WordNet synset più rappresentativi per il concetto in input, quelli relativi agli iponimi che hanno overlap maggiore con il contesto generale delle definizioni.

```
def getAllHyponyms(word):
    hypo_list = []
    for ss in wn.synsets(word): #for all the WN synsets of the word (the input genus)
        hypo_list.extend(ss.hyponyms()) #extend the hyponim list with all the WN hyponim synsets retrieved
    return hypo_list

def getSynsetsOverlap(synsets, context):
    """
    Create a list of tuples that contains all the synsets and the corrispective overlap with the given context
    Input:
        synsets = lists of synsets (of all the hyponims retrieved from the genus/hyperonim considered)
        context = list of processed words from the definitions of every concept
    Output:
        best_synsets =  list of tuples in the form : [(synset 1, overlap 1),...(synset n, overlap n)]
    """
    best_synsets = []
    for syn in synsets: #for each of the hyponim synsets passed as parameters
        syn_context = getSynsetContext(syn)
        #comparing the context of a given hyponim synset and the global context (of the complete list of the processed definition)
        overlap = len(set(syn_context) & set(context))
        best_synsets.append((syn, overlap))
    return best_synsets # list of tuples (synset, overlap)
```

Abbiamo usufruito, pertanto, nella nostra ricerca del senso più appropriato di un concetto, partendo da una serie di definizioni di tale concetto, del “Genus-differentia mechanism”, che recita:
“ per descrivere un concetto le cose fondamentali sono 2:
- Genus: per descrivere un concetto dobbiamo inserirlo all'interno di una tassonomia, circoscrivere un raggio d'azione per quel concetto. Es: dico che la banana è un frutto;
- Differentia: tutto quello che caratterizza quel concetto in maniera differenziale da tutti gli altri concetti (la banana è gialla, di forma allungata, ecc.). “

- Risultati: in output restituiamo una tabella, tramite il modulo “PrettyTable”, rappresentante, per ogni riga, il vero concetto, quello restituito dalla computazione automatica (il synset associato al miglior iponimo trovato) e la definizione su WordNet di quest’ultimo.
  Nel file .txt “allFoundConcept” mostriamo, inoltre, i primi 10 synset più rappresentativi per ognuno dei 4 concetti oggetto di studio.

Correct Concept	Best WordNet Synset Found	Definition
courage	physical_ability.n.01	the ability to perform some physical act; contrasting with mental ability
paper	composite_material.n.01	strong lightweight material developed in the laboratory; fibers of more than one kind are bonded together chemically
apprehension	apprehension.n.01	fearful expectation or anticipation
sharpener	drill.n.01	a tool with a sharp point and cutting edges for making holes in hard materials (usually rotating rapidly or by repeated blows)

Result output also foundable at this path: /Users/itsallmacman/Downloads/Di Caro 2/Esercizio 1.5/output/bestConcepts.txt

```
courage
[Synset('physical_ability.n.01'), Synset('stage_fright.n.01'), Synset('take_the_bull_by_the_horns.v.01'), Synset('form.n.14'),

paper
[Synset('composite_material.n.01'), Synset('paper.n.01'), Synset('literature.n.03'), Synset('aggregate.n.02'), Synset('bimetal

apprehension
[Synset('apprehension.n.01'), Synset('panic.n.01'), Synset('panic.n.02'), Synset('edginess.n.01'), Synset('insecurity.n.02'),

sharpener
[Synset('drill.n.01'), Synset('jaws_of_life.n.01'), Synset('plow.n.01'), Synset('upset.n.04'), Synset('lead_pencil.n.01'), Syn
```

Esercizio 2.1

- Obiettivo: implementare un semplice algoritmo di text summarization di tipo estrattivo (scelta della seconda opzione).
  Esso consiste, dato un documento in input originale, nell’ottenere in output un documento ridotto di una percentuale a nostra scelta mantenendo il più possibile il contenuto semantico generale.
  In particolare, i metodi estrattivi, consistono appunto nell’estrarre le frasi di maggior rilevanza per l’intero testo, mantenendole per il riassunto risultante.
  Al contrario, vi sono anche dei metodi astrattivi, che usano parafrasi e metodi di ri-generazione del testo per rielaborarlo, ottenendo nuove frasi non presenti nel documento iniziale.

- Implementazione:

  1. Scelta di un’euristica/criterio di rilevanza per effettuare il ranking delle frasi nel documento originale: Posizione all’interno del testo, Metodo del titolo, Optimum Position Policy (OPP), Cue Phrases.
     Di queste abbiamo scelto di usare il metodo del titolo, ossia l’idea secondo cui le keywords presenti nel titolo aiutino a trovare i contenuti (le frasi) più rilevanti, ovvero quelle nelle quali tali keywords sono maggiormente presenti.

2. Al fine di effettuare tali match, abbiamo usato una rappresentazione vettoriale sia per il topic (le keywords del titolo) che per le parole contenute nei vari paragrafi del testo: i vettori NASARI.
3. Una volta ottenute le rappresentazioni vettoriali del contesto a cui far riferimento (in questo caso il titolo) e dei vari paragrafi del testo, effettuiamo un calcolo di similarità tra i termini rappresentati da tali vettori.
   A tal fine, utilizziamo la metrica del Weighted Overlap.

Then, the square-rooted Weighted Overlap (Pilehvar et al., 2013) as vector comparison method can be used,

$$WO(v_1, v_2) = \frac{\sum_{q \in O} \left(rank(q, v_1) + rank(q, v_2)\right)^{-1}}{\sum_{i=1}^{|O|} (2i)^{-1}}$$

where O is the set of overlapping dimensions between the two vectors and $rank(q, v_i)$ is the rank of dimension q in the vector v_i.
The similarity between words w_1 and w_2 is computed as the similarity of their closest senses

$$sim(w_1, w_2) = \max_{v_1 \in \mathcal{C}_{w_1}, v_2 \in \mathcal{C}_{w_2}} \sqrt{WO(v_1, v_2)}$$

4. Infine, riordiniamo le frasi mantenute dopo il calcolo della similarità (quelle con WO più alto), al fine di mantenere l'ordine dei paragrafi del documento originale, e scegliamo una soglia limite (in percentuale rispetto all'intero documento) oltre la quale escludere i paragrafi dal riassunto, secondo i valori precedenti di Weighted Overlap calcolati.

```
# getting the topics based on the document's title.
topics = get_title_topic(document, nasari_dict)

paragraphs = []
i = 0
# for each paragraph, except the title (document[0])
for paragraph in document[1:]:
    context = create_context(paragraph, nasari_dict)
    paragraph_wo = 0  # Weighted Overlap average inside the paragraph.

    for word in context:
        # Computing WO for each word inside the paragraph.
        topic_wo = 0
        for vector in topics:
            topic_wo = topic_wo + weighted_overlap(vector, word)
        if topic_wo != 0:
            topic_wo = topic_wo / len(topics)

        # Sum all words WO in the paragraph's WO
        paragraph_wo += topic_wo

    if len(context) > 0:
        paragraph_wo = paragraph_wo / len(context)
        # append in paragraphs a tuple with the index of the paragraph (to
        # preserve order), the WO of the paragraph and the paragraph's text.
        paragraphs.append((i, paragraph_wo, paragraph))
    i += 1

to_keep = len(paragraphs) - int(round((percentage / 100) * len(paragraphs), 0))

# Sort by highest score and keeps all the important entries. From first to "to_keep"
new_document = sorted(paragraphs, key=lambda x: x[1], reverse=True)[:to_keep]
# Restore the original order.
new_document = sorted(new_document, key=lambda x: x[0], reverse=False)
# delete unnecessary fields (x[0] which contains the "i" and x[1] which
# contains the WO of the paragraph) inside new_document in order to
# keep oly the text (I associated to each paragraph a score based on the
# "importance").
new_document = list(map(lambda x: x[2], new_document))

new_document = [document[0]] + new_document # title + paragraphs
return new_document
```

- Risultati: al fine della valutazione dei riassunti ottenuti (quanto sono appropriati per rappresentare il contenuto semantico dei documenti originali) utilizziamo due misure ben note in questo campo: BLEU (bilingual evaluation understudy), riguardante la precision, e ROUGE (Recall-Oriented Understudy for Gisting Evaluation) rappresentante, invece, la recall.
  Scegliamo, pertanto, di confrontare i nostri riassunti e le parole in essi risultanti dall'algoritmo implementato con le parole che, per ogni documento originale, hanno uno score maggiore di tf-idf (Term frequency - inverse document frequency) e, pertanto, secondo tale indice di rilevanza, sarebbero quelle più indicate come "da mantenere" nel riassunto estratto.

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|} \qquad \text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

Esercizio 2.2

- Obiettivo: implementare un algoritmo di topic modelling.
  Siamo nell'ambito della semantica documentale.
  Data una collezione, anche molto vasta, di documenti trovarne i temi principali ("topic").
  Quello del topic modelling è un modello statistico e probabilistico non supervisionato, in quanto non necessita di dati annotati.
  Un topic è, pertanto, una lista pesata di parole (ordinate in base all'importanza delle parole stesse) estratta dal documento (collezione di documenti) originale.

- Implementazione (utilizzando la libreria Gensim):

  1. Abbiamo reperito da Kaggle il corpus "Medium Data Science Articles Topic Modelling", contenente informazioni su molteplici articoli della testata online "Medium" riguardanti tutti gli ambiti della Data-Science, da cui estrarne 10 topic principali.

  2. Fase di pre-processing dei dati (del testo contenuto nel corpus scelto)

  3. Creazione di un dizionario dai dati di testo processati e successiva conversione in un corpus in formato BagOfWords.
     Inoltre, effettuiamo il salvataggio di tali risultati per eventuali utilizzi futuri.

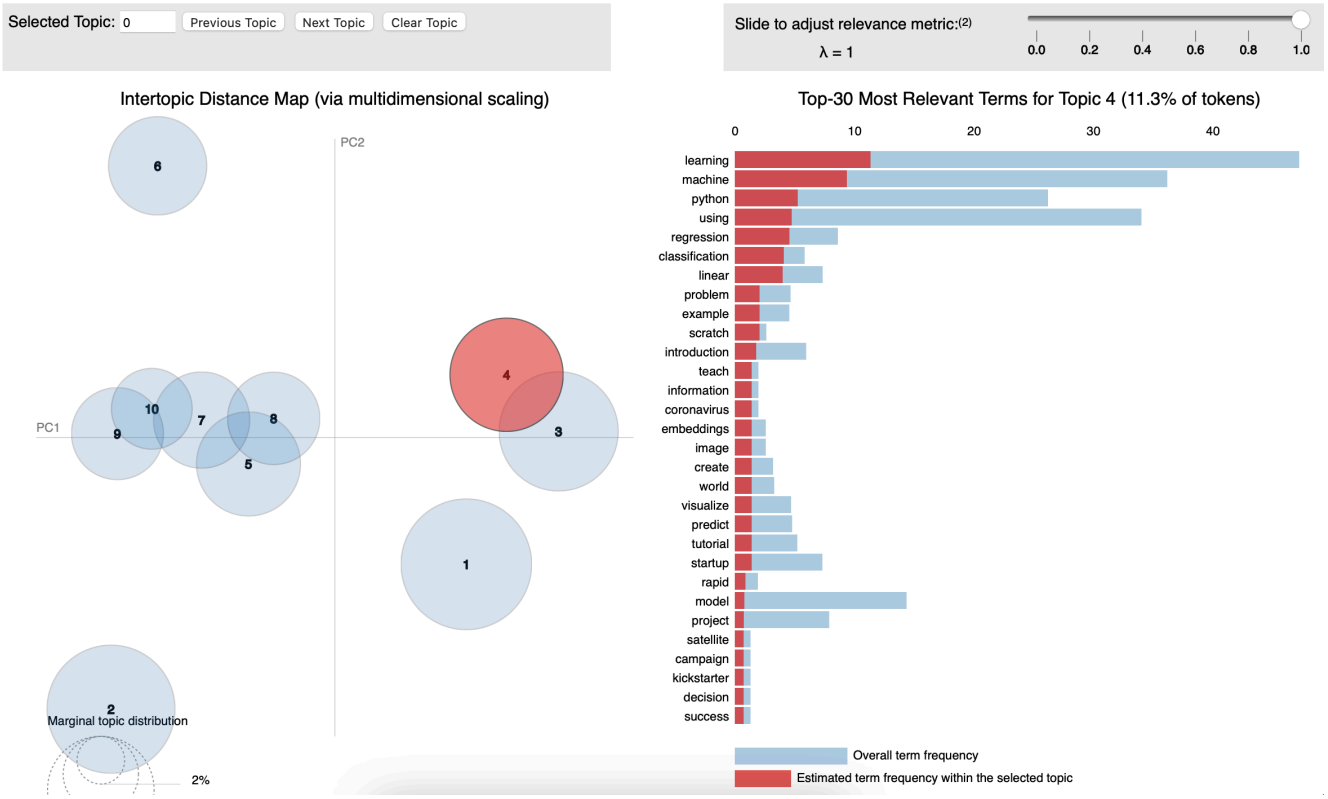
```
#First, we are creating a dictionary from the data, then convert to bag-of-words corpus
#and save the dictionary and corpus for future use.
dictionary = corpora.Dictionary(text_data)
corpus = [dictionary.doc2bow(text) for text in text_data]
pickle.dump(corpus, open(path + '/result/' + 'corpus.pkl', 'wb'))
dictionary.save(path + '/result/' + 'dictionary.gensim')
```

  4. Giunti a questo punto, utilizziamo la tecnica della Latent Dirichlet Allocation (LDA), consistente nella versione probabilistica della LSA (e perciò detta anche "pLSA"), che sfrutta la statistica bayesiana.
     Si basa sull'assunzione che un documento è un mix di topics e ogni parola ha una certa probabilità di comparsa in ogni topic.
     Tramite il metodo LdaModel(), pertanto, ricaviamo il numero di topic richiesto.
     Scegliamo di estrarne 10 in quanto risulta il numero indicato come appropriato per il dataset da noi utilizzato.

- Risultati:
  1. Ciò che otteniamo è una lista pesata (in base all'importanza delle parole stesse) di parole estratte dal corpus, rappresentanti per ciascuno dei 10 topic individuati, le 3 parole (ovvero le 3 topic-words) più rappresentative, in quanto quelle di maggior significatività per l'algoritmo di LDA.

```
Analyzing 37332 titles..
(0, '0.044*"python" + 0.020*"panda" + 0.015*"guide"')
(1, '0.051*"learning" + 0.051*"using" + 0.049*"machine"')
(2, '0.020*"science" + 0.016*"model" + 0.014*"guide"')
(3, '0.053*"learning" + 0.035*"analytics" + 0.034*"machine"')
(4, '0.015*"simple" + 0.015*"scientist" + 0.011*"statistics"')
(5, '0.015*"science" + 0.015*"analytics" + 0.010*"getting"')
(6, '0.049*"learning" + 0.041*"machine" + 0.023*"python"')
(7, '0.013*"machine" + 0.013*"analytics" + 0.009*"vidhya"')
(8, '0.089*"science" + 0.014*"using" + 0.013*"python"')
(9, '0.026*"science" + 0.016*"learning" + 0.012*"analysis"')
```

  2. Infine, visualizziamo i risultati riguardanti i topic ottenuti e le parole di essi maggiormente rappresentativi, tramite il modulo. "pyLDAvis", ed in particolare il metodo prepare(), che ci permette di trasformare and preparare i dati di un modello LDA per la loro visualizzazione. Scegliamo, inoltre, di salvare e successivamente, in modo automatico al termine dell'esecuzione (tramite il metodo open() del modulo webbrowser), aprire un Html contenente i topic ottenuti, con annesse parole più rappresentative.



Esercizio 3.1

- Obiettivo: partendo da un qualsiasi tipo di dato testuale, costruire un grafo (Knowledge graph) e implementare un semplice meccanismo per visualizzare i dati testuali, trovare le giuste informazioni (IR, disambiguazione).
  Un Knowledge graph assume importanza quando si guarda a dati che hanno una certa rilevanza quando messi in relazione, ed ha una struttura a grafo tipizzato con proprietà.
  Facendo l'esempio di uno dei dei più noti Knowledge Graph, Neo4j, esso presenta una struttura con delle proprietà chiave-valore, label e type e direzionamento delle relazioni.

- Implementazione (utilizzando Neo4j, un software per basi di dati a grafo open source sviluppato interamente in Java):

  1. Connessione in localhost col database creato (e preventivamente aperto) su Neo4j

  2. Rimozione di tutti gli eventuali nodi (le etichette associate) e le relazioni precedentemente presenti

  3. Popolazione del database (inserendo le etichette dei nodi e relazioni desiderate): per far ciò prendiamo i dati di nostro interesse dal dataset utilizzato (nel nostro caso "euro2020.csv", rappresentante varie informazioni sui principali top player dell'ultima competizione europea per nazioni).
     Una volta reperiti tali dati, procediamo col riempimento di un dizionario "parsed_dict", contenente le informazioni su tutti i nostri nodi creati e, analogamente, ci occupiamo anche delle relationships, eseguendo per ognuna di esse il metodo run() (sulla sessione attuale di connessione).

```
#storing all informations relative to all the nodes created
parsed_dict ={'Country':Country, 'Player':Player, 'Goals':Goals, 'Matchplayed':MatchPlayed, 'Position':Position, 'Distancecovered':DistanceCovered}

# run first relation with attached informations (parsed_dict)
session.run(Country_of,parsed_dict)
# run second relation with attached informations (parsed_dict)
session.run(GoalsPerMatch, parsed_dict)
# run third relation with attached informations (parsed_dict)
session.run(KmCovered_as_Role, parsed_dict)
```

```
# taking all the nodes of interest from the dataset
Country = df["Country"].unique()

Player = df["Player"].unique()

Goals = df["Goals"].unique()

MatchPlayed = df["Matchplayed"].unique()

Position = df["Position"].unique()

DistanceCovered = df["Distancecovered"].unique()

#creation of the first relationship: A = Player, B = Country -> A plays_for B)
Country_of = '''MERGE(A:Player{Player:$Player})
    MERGE(B:Country{Country:$Country})
    MERGE (A)-[:plays_for]->(B)
    '''
#creation of the second relationship: A = Player, B = Goals, C = MatchPlayed ->
GoalsPerMatch = '''MERGE(A:Player{Player:$Player})
    MERGE(B:Goals{Goals:$Goals})
    MERGE(C:MatchPlayed{MatchPlayed:$Matchplayed})
    MERGE (C)<-[:in_matches_number]-(A)-[:scored_goals_number]->(B)
```

4. Esecuzione delle query preparate sui dati caricati

```
#bonus query to show the 10 players who scored the most goals in EURO2020, viewing also how many matches they've played
querybonus = graph.run("MATCH (matches)<-[r1:in_matches_number]-(player)-[r:scored_goals_number]->(goals) RETURN player,goals,matches ORDER BY goals.Goals DESC LIMIT 10
print("< Euro2020 - Players ranked by their goals scored: >" + "\n")
for elem in querybonus:
    print(f"Player: {elem['player']['Player']} has scored tot number goals: {elem['goals']['Goals']} in tot number matches played: {elem['matches']['MatchPlayed']}")
```

- Risultati:

1. Su terminale abbiamo in output i risultati delle varie queries

```
< Euro2020 - Players ranked by their goals scored: >

Player: Patrik Schick has scored tot number goals: 5 in tot number matches played: 5
Player: Cristiano Ronaldo has scored tot number goals: 5 in tot number matches played: 4
Player: Harry Kane has scored tot number goals: 4 in tot number matches played: 7
Player: Kasper Dolberg has scored tot number goals: 4 in tot number matches played: 4
Player: Emil Forsberg has scored tot number goals: 4 in tot number matches played: 4
Player: Romelu Lukaku has scored tot number goals: 4 in tot number matches played: 5
Player: Karim Benzema has scored tot number goals: 4 in tot number matches played: 4
Player: Raheem Sterling has scored tot number goals: 3 in tot number matches played: 7
Player: Xherdan Shaqiri has scored tot number goals: 3 in tot number matches played: 5
Player: Robert Lewandowski has scored tot number goals: 3 in tot number matches played: 3
```

2. Sulla sezione Browser / Bloom dell'applicativo Neo4j Desktop possiamo visualizzare ogni possibile grafico ricavabile dalla combinazione di nodi e relazioni di nostro piacimento, utilizzando queries analoghe alle precedenti.
Riportiamo qui di seguito una serie di risultati a titolo di esempio, estratti dalla visualizzazione dei grafici risultanti da una serie di queries eseguite su Neo4j Browser.

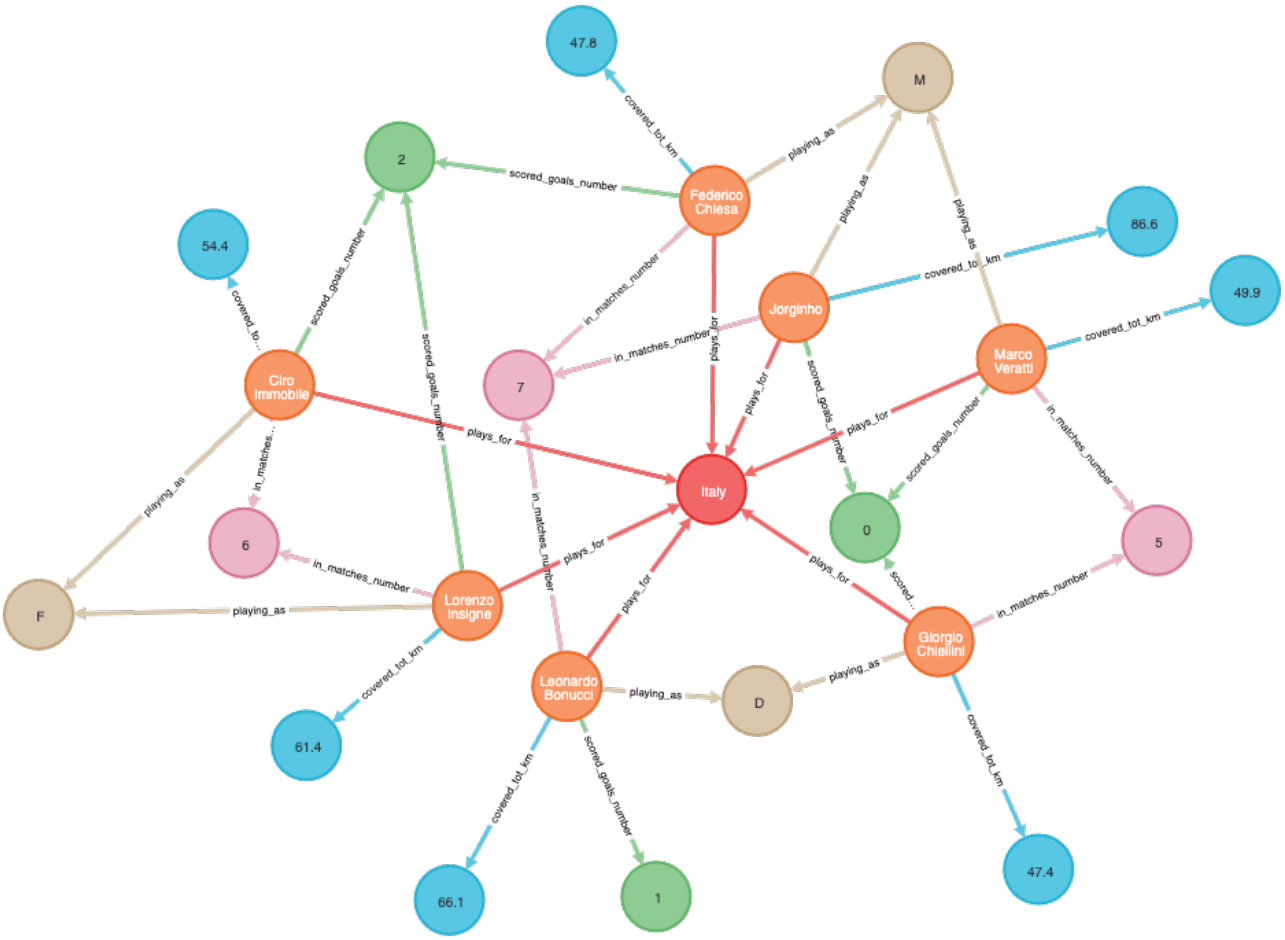
1. Giocatori che giocano per la nazione "Italia" e che hanno come ruolo una qualsiasi posizione; scegliamo di vedere in output i nodi risultanti "player", "position" e "country", e tutte le relazioni fra essi instaurate (oltre a quelle citate nella query "playing_as" e "playing_as" sono visibili anche "scored_goals_number", "covered_tot_km", "in_matches_number").

2. Uguale alla precedente ma qui, come esplicitamente richiesto nel campo RETURN della query, vediamo in output, oltre alle node labels, solo le due relazioni indicate.

3. Giocatori che giocano per una qualunque nazione e che hanno come ruolo la posizione di attaccante ("F", che sta per "forward"); richiediamo di visualizzare le informazioni risultati riguardanti i nodi "player", "position"(limitata ai filler "F") e "country".

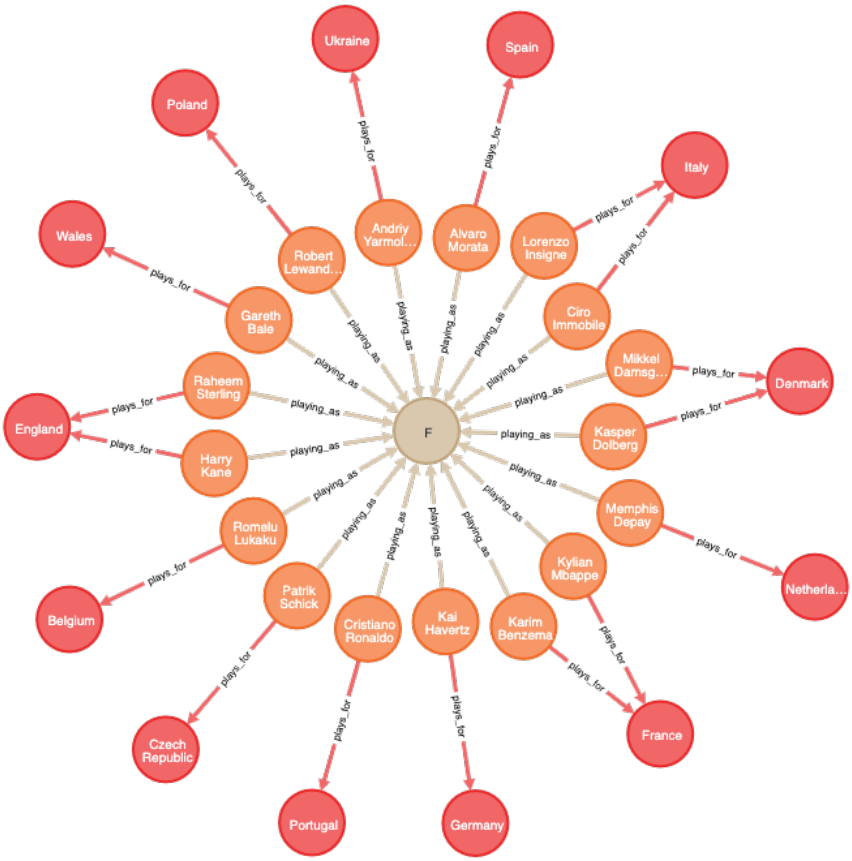
4. Giocatori che hanno come ruolo una qualsiasi posizione e che hanno segnato un certo (qualsiasi) numero di gol complessivi nella competizione; visualizziamo le informazioni risultanti che riguardano i nodi "player", "role" e "goals".

MATCH (position)<-[r1:playing_as]-(player)-[r:plays_for]->(country{Country:'Italy'}) RETURN player,position,country LIMIT 25



MATCH (position)<-[r1:playing_as]-(player)-[r:plays_for]->(country{Country:'Italy'}) RETURN player,position,country,r1,r LIMIT 25

```
MATCH (position{Position:'F'})<-[r1:playing_as]-(player)-[r:plays_for]->(country) RETURN player,country,position
```




```
MATCH (goals)<-[r1:scored_goals_number]-(player)-[r:playing_as]->(role) RETURN player,role,goals LIMIT 50
```