IST 2016
Processamento e Recuperação de Informação
Project Report - Part 1 - Group 4

1 - Simple approach based on TF-IDF

Começamos por ler um english textual document (parte de um capítulo de Alice In Wonderland) onde filtramos as palavras. Tiramos os símbolos de pontuação, menos o apóstrofo (para manter palavras como “don’t” intactas), e de seguida as stopwords. A seguir calculamos bigramas através dessas palavras e juntamos às mesmas. Assim ficamos com uma lista de candidatos com todas as palavras e bigramas.

Depois vamos buscar ao 20newsgroups o train.data e juntamos a esse conjunto os candidatos recolhidos anteriormente. Desta maneira, ao calcular o IDF, todos os termos do documento que fornecemos existem no vocabulário, permitindo que haja smoothing ao IDF.

De seguida calculamos o TF do english textual document e calculamos o TF-IDF. Em seguida, ordenamos os resultados por ordem crescente e selecionamos os 5 valores maiores.

2 - Evaluating the simple approach

O primeiro passo é a leitura dos 30 documentos do FAO30 e das suas keys pelas várias equipas, juntando-as para cada documento (utilizando Python Sets para excluir repetições).
De seguida, utilizando um TFIDFVectorizer, obtemos os TF-IDF para cada par documento-termo.
Para cada documento, essas pontuações por termo são usadas para reordenar crescentemente uma lista de índices, que correspondem às posições de cada termo no vocabulário do Vectorizer. Depois podemos obter os cinco termos com maior índice, por documento, pois também terão os valores TF-IDF mais altos.

Para as medidas de desempenho, para cada documento, as listas de termos relevantes e as de termos que o programa indica como relevantes são comparadas.
A average-precision é calculada em 5 iterações, correspondendo aos 5 termos obtidos para cada documento.
Comparando com as classificações feitas pelas equipas, os nossos resultados são fracos, com precision, recall, F1, AP e mean-average-precision baixos. Isto deve-se a, por vezes, entre os termos relevantes indicados pelas equipas e pelo programa haver somente um resultado em comum. Esta “simple approach” não é, portanto, a indicada.

3 - Improving the simple approach

O primeiro passo neste exercício foi reescrever a gramática do enunciado de modo a ser legível pelo pacote “re”. Uma das alterações que tivemos de fazer foi acrescentar “$” no fim da gramática de modo a que, quando um termo fizesse match à expressão, fosse rejeitasse

caso a seguir houvesse mais algum símbolo. Acrescentámos também “[A-Z]*” no lugar do “.*” para permitir que fosse possível fazer match com qualquer letra dentro da tag “<NN>”, como por exemplo “<NNP>” para nomes próprios.

Após a reescrita da gramática é-nos possível filtrar os n-grams que definimos como candidatos. De seguida, calculamos as frequências dos termos por documento usando um CountVectorizer, usando um tokenizer custom[1] que faz a verificação pela expressão regular.

Em seguida, calculamos o IDF segundo a fórmula do enunciado, depois usado na fórmula do BM25. O seu parâmetro f(t,D) é obtido na matriz esparsa criada pelo Vectorizer.
O ordenamento dos candidatos é feito como nos exercícios anteriores, retornando também 5 resultados por documento.

4 - A more sophisticated approach

Este programa utiliza parte da lógica presente no que foi feito para a questão 3. Utiliza-se um CountVectorizer com um tokenizer “custom” que, desta vez, não verifica os termos segundo uma expressão regular mas de novo separa as palavras em bigramas e trigramas e mantém mais algumas estatísticas sobre os termos eos documentos. Por exemplo, é calculada a soma das frequências de cada termo em todos os documentos, para ser utilizado no cálculo das probabilidades.

Para o cálculo da informativeness, em que se usam Language Models apenas com unigramas (para P(W) e Q(W)), há independência entre as palavras dentro de termos. Por isso, a probabilidade de ocorrência de um termo é apenas o produto das probabilidades das palavras que o constituem. Consideramos unigramas, bigramas e trigramas como possíveis termos e limitamos o cálculo da probabilidade de ocorrência de sequências a três produtos, ou seja, depender apenas de um número reduzido de palavras anteriores (N = 3).

Para o cálculo da phraseness, utilizamos uma aproximação em que reduzimos o número de cálculos, “cortando” numeradores e denominadores entre as parcelas do produtório usado no cálculo da probabilidade. Este produtório torna-se apenas uma divisão, da seguinte forma:

$$P(W) \approx \prod_{i=1}^{m} \frac{count(w_{i-(N-1)},\dots,w_i)}{count(w_{i-(N-1)},\dots,w_i)} = \frac{count(w_1)}{count(all)} \frac{count(w_1,w_2)}{count(w_1)} \frac{count(w_1,w_2,w_3)}{count(w_1,w_2)} = \frac{count(W)}{count(all)}$$

Nos cálculos envolvendo probabilidades em foreground e background documents, as contagens utilizadas são feitas apenas no documento em avaliação ou em todo o corpus (através de valores acumulados guardados nas estatísticas), respectivamente.

[1] http://scikit-learn.org/stable/modules/feature_extraction.html#customizing-the-vectorizer-classes

A adição de informativeness e phraseness para cada par termo-documento é guardada e ordenada como descrito nas questões anteriores, produzindo, mais uma vez, cinco termos mais relevantes para cada documento.

Solution - Exercise 1

```
import nltk
import string
from nltk.tokenize import RegexpTokenizer
from nltk.corpus import stopwords
from nltk.collocations import *
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
import operator

punct = string.punctuation
punct = punct.translate(None,"'")
exclude = set(punct)
stop = set(stopwords.words('english'))

def test_set(s):
    return ''.join(ch for ch in s if ch not in exclude)

#exercise 1 - part 1 - reading words
#english textual document
etd = open('englishtextualdoc.txt')
etdread = etd.read()
etdread = etdread.decode('utf-8')
etdsentences = nltk.sent_tokenize(etdread)
etdwords = list()
for etdsentence in etdsentences:
    sentencenopunct = test_set(etdsentence.lower())
    sentencewords = nltk.word_tokenize(sentencenopunct)
    for i in range(len(sentencewords)):
        word = sentencewords[i]
        if word[0] == "'":
            word = word[1:]
        sentencewords[i] = word
    sentenceclean = [i for i in sentencewords if i not in stop]
    etdwords += sentenceclean
etdbigrams = list(nltk.bigrams(etdwords))
#print etdbigrams
candidates = list()
candidates += etdwords
for bi in etdbigrams:
    candidates += [bi[0]+" "+bi[1]]


train = fetch_20newsgroups(subset='train')
englishdocplustrain = train.data + candidates
test = fetch_20newsgroups(subset='test')
vectorizer2 = TfidfVectorizer( use_idf=True, ngram_range=(1,2))
trainvec2 = vectorizer2.fit_transform(englishdocplustrain)


globalwords = {}
DFdict = {}
```

```
def processDoc(docwords, name):
    f1counts = {}
    for word in docwords:
        if word in f1counts.keys():
            f1counts[word]+=1
        else:
            f1counts[word] = 1

    for word in f1counts.keys():
        if word in globalwords.keys():
            globalwords[word][name] = f1counts[word]
        else:
            globalwords[word] = {}
            globalwords[word][name] = f1counts[word]
    return


processDoc(candidates, "Alice")
vec3vocab = vectorizer2.vocabulary_
idfdict = {}
for term in candidates:
    try:
        idfdict[term] = vectorizer2.idf_[vec3vocab[term]] *
globalwords[term]['Alice']
    except Exception:
        pass

sorted_x = sorted(idfdict.items(), key=operator.itemgetter(1))
for i in range(5):
    print sorted_x[i]
```

Solution - Exercise 2

```
import nltk
import string
from nltk.tokenize import RegexpTokenizer
from nltk.corpus import stopwords
from nltk.collocations import *
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.metrics import f1_score

import operator
import numpy

#quicksort
def sort(array):
    less = []
    equal = []
    greater = []

    if len(array) > 1:
```

```
        pivot = array[0]
        for x in array:
            if x < pivot:
                less.append(x)
            if x == pivot:
                equal.append(x)
            if x > pivot:
                greater.append(x)
        return sort(less)+equal+sort(greater)
    else:
        return array

#import 30 documents
import os
path = "fao30/documents/"

all_docs = []
docindexnames = dict()
docindex = 0
for filename in os.listdir(path):
    docindexnames[docindex] = filename
    etd = open(path + filename)
    etdread = etd.read()
    etdread = etdread.decode('latin-1')
    etd_words = nltk.word_tokenize(etdread)
    all_docs += [etdread]
    docindex += 1

stop = set(stopwords.words('english'))

#build tk-idf array with unigrams and bigrams and exclude stop_words
vectorizer2 = TfidfVectorizer( use_idf=True, ngram_range=(1,2), stop_words=stop )
docstfidf = vectorizer2.fit_transform(all_docs)
vecvocab = vectorizer2.vocabulary_


####################
#get all relevants and merge the relevants for the same documents
path = "fao30/indexers/iic1/"
keysfordoc = dict()
indexerIterator = 1
setForKeys = set()
for filename in os.listdir(path):
    while indexerIterator <= 6:
        etd = open(path + filename)
        etdread = etd.read()
        etdread = etdread.decode('latin-1')
        etd_keys = etdread.split("\n",-1)[:-1]
        fname = filename[:-4] + '.txt'

        setForKeys = setForKeys.union(set(etd_keys))
```

```
        etd_keys = list(setForKeys)
        keysfordoc[fname] = etd_keys
        indexerIterator += 1
        path = path[:-2] + str(indexerIterator) + "/"
    path = "fao30/indexers/iic1/"
    indexerIterator = 1


#calculate tf-idf for each document
doccandidateslist = dict()
featurenames = list(vectorizer2.get_feature_names())
for idoc in range(len(docindexnames)):
    docname = docindexnames[idoc]
    featurenamescopy = numpy.array(featurenames)
    tfidfdoccopy = numpy.array(docstfidf.getrow(idoc).toarray()[0])
    sortedindices = (tfidfdoccopy.argsort()[-5:])[::-1]
    candidatewordsfordoc = list()
    for candidatei in sortedindices:
        candidatewordsfordoc += [featurenamescopy[candidatei]]
    doccandidateslist[docname] = candidatewordsfordoc



#calculate precision, reccall and f1
measuresdoc = dict()
for idoc in range(len(docindexnames)):
    measures= dict()
    docname = docindexnames[idoc]
    setrelevant = set(keysfordoc[docname])
    setanswer = set(doccandidateslist[docname])
    sizeRel = len(setrelevant)
    sizeAns = len(setanswer)
    sizeInt = len(setrelevant.intersection(setanswer))
    pr = sizeInt/(0.0+sizeAns)
    re = sizeInt/(0.0+sizeRel)
    try:
        f1 = (2*re*pr)/(re+pr)
    except ZeroDivisionError:
        f1 = 0

    measures["pr"] = pr
    measures["re"] = re
    measures["f1"] = f1
    measuresdoc[docname] = measures
#calculate AP and therefore MAP
aptotalsum = 0
for idoc in range(len(docindexnames)):
    docname = docindexnames[idoc]
    setrelevant = set(keysfordoc[docname])
    setanswer = set(doccandidateslist[docname])
```

```
    sizeRel = len(setrelevant)
    sizeAns = len(setanswer)
    sizeInt = len(setrelevant.intersection(setanswer))
    pr = sizeInt/(0.0+sizeAns)
    re = sizeInt/(0.0+sizeRel)
    listansweri = []
    setansweri = set()
    apitersum = 0
    for term in range(len(doccandidateslist[docname])):
        listansweri += [doccandidateslist[docname][term]]
        setanseri = set(listansweri)
        p = (len(setanseri.intersection(setrelevant)))/(len(setanseri)+0.0)
        r = int()
        if doccandidateslist[docname][term] in keysfordoc[docname]:
            r = 1
        else:
            r = 0
        apitersum += (p * r)
    ap = apitersum  / (sizeRel+0.0)
    measuresdoc[docname]["ap"] = ap
    aptotalsum += ap
    listanswer = []
    setansweri = set()


print measuresdoc
print "map: " + str(aptotalsum/len(docindexnames))
```

Solution - Exercise 3

```
import nltk
import string
from nltk.tokenize import RegexpTokenizer
from nltk.corpus import stopwords
from nltk.collocations import *
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
import operator
import re
from math import log10
import numpy


#general statistics that we'll calculate further down
numberofdocuments = 0
totalwordsperdocument = dict()
totalwordsincorpus = 0
```

```
#checking if a term (uni/bi/trigram) matches a regex.
#conversao passo-a-passo da gramatica do enunciado
#grammar2 = r'(<NN[A-Z]*>)+$'
#grammar2 = r'(<JJ> <NN[A-Z]*>)+$'
#grammar2 = r'((<JJ>)* <NN[A-Z]*>)+$'
#grammar2 = r'((<IN>)?(<JJ>)*<NN[A-Z]*>)+$'
#grammar2 = r'((<IN>)?(<JJ>)*(<NN[A-Z]*>)+)+$'
grammar2 = r'(((<JJ>)*(<NN[A-Z]*>)+<IN>)?(<JJ>)*(<NN[A-Z]*>)+)+$'
regexparser2 = re.compile(grammar2)
def filterCandidates(candidatesList):
    newCandidates = []
    for candidate in candidatesList:
        stringtocheck = ""
        if isinstance(candidate, tuple):
            candidate = " ".join(candidate)
        candidatetags = nltk.pos_tag(nltk.word_tokenize(candidate))
        for taggedterm in candidatetags:
            stringtocheck += "<"+taggedterm[1]+">"
        parseResult = regexparser2.match(stringtocheck)
        if parseResult:
            newCandidates += [candidate]
    return newCandidates


#CUSTOM TOKENIZER
#this tokenizer will split words, delete stop words and punctuation and
#transform it into uni/bi/trigrams, which will all be filtered by the regex.
#some document and general statistics are also calculated here.
punct = string.punctuation
punct = punct.translate(None,"'")
punctexcludeset = set(punct)
stop = set(stopwords.words('english'))
docindex = 0
def test_set(s):
    return ''.join(ch for ch in s if ch not in punctexcludeset)
def my_tokenizer(documentasstring):
    #these variables have to be declared here as global so the vectorizer can
    #see them from our program (we first declared them earlier)
    global docindex
    global totalwordsincorpus
    global numberofdocuments
    docsentences = nltk.sent_tokenize(documentasstring)
    docwords = list()
    doclength = 0
    for docsentence in docsentences:
        sentencenopunct = test_set(docsentence.lower())
        sentencewords = nltk.word_tokenize(sentencenopunct)
        doclength += len(sentencewords)
        for i in range(len(sentencewords)):
            word = sentencewords[i]
            if word[0] == "'":
```

```
                word = word[1:]
            sentencewords[i] = word
        sentenceclean = [i for i in sentencewords if i not in stop]
        docwords += sentenceclean
    totalwordsperdocument[docindex] = doclength
    totalwordsincorpus += doclength
    #all the words are split
    docbigrams = list(nltk.bigrams(docwords))
    doctrigrams = list(nltk.trigrams(docwords))
    docterms = docwords + docbigrams + doctrigrams
    docvalidterms = filterCandidates(docterms)
    docindex +=1
    numberofdocuments +=1
    return docvalidterms

#PROCESSING:

#documents from FAO30 (same as exercise-2)

import os
path = "fao30/documents/"

all_docs = []
docindexnames = dict()
docreadindex = 0
for filename in os.listdir(path):
    docindexnames[docreadindex] = filename
    etd = open(path + filename)
    etdread = etd.read()
    etdread = etdread.decode('latin-1')
    etd_words = nltk.word_tokenize(etdread)
    all_docs += [etdread]
    docreadindex += 1


#uses a vectorizer to calculate term frequency
countVectorizer = CountVectorizer(tokenizer=my_tokenizer)
countVectorizer.build_analyzer()
#docstf = countVectorizer.fit_transform(set(["Alice stopped by the big big station
to retrieve the blue poop","Alice stopped by a poop and was angry"]))
docstf = countVectorizer.fit_transform(all_docs)
vecvocab = countVectorizer.vocabulary_


#calculates the IDF according to the new formula
idfDict = dict()
for termi in range(len(vecvocab)):
    docswithterm = docstf.getcol(termi).getnnz(0)[0]
    numerator = numberofdocuments - docswithterm +0.5
    denominator = docswithterm +0.5
    idfDict[termi] = log10(numerator/denominator)
```

```
#calculates the BM25 for a term,document
k1 = 1.2
b = 0.75
def score(documentn, termi):
    idfpart = idfDict[termi]
    ftD = docstf[documentn,termi]
    avgdl = totalwordsincorpus/(0.0+numberofdocuments)
    #print ftD
    numeratorpart = ftD * (k1 + 1)
    denominatorpart = ftD + k1 * (1 - b + b *
(totalwordsperdocument[documentn]/avgdl) )
    scoredt = idfpart * (numeratorpart / denominatorpart)
    return scoredt


#all the scores will be calculated here
dictscores = dict()
for doci in range(numberofdocuments):
    documentscores = dict()
    for termi in range(len(vecvocab)):
        documentscores[termi] = score(doci, termi)
    dictscores[doci] = documentscores


doccandidateslist = dict()
featurenames = list(countVectorizer.get_feature_names())
featurenamescopy = numpy.array(featurenames)
for idoc in range(numberofdocuments):
    bm25doccopy = numpy.array(dictscores[idoc].values())
    #the keys were inserted by order, so the values were by this order as well
    sortedindices = (bm25doccopy.argsort()[-5:])[::-1]
    candidatewordsfordoc = list()
    for candidatei in sortedindices:
        candidatewordsfordoc += [featurenamescopy[candidatei]]
    doccandidateslist[idoc] = candidatewordsfordoc

print doccandidateslist
```

Solution - Exercise 4

```
import nltk
import string
from nltk.tokenize import RegexpTokenizer
from nltk.corpus import stopwords
from nltk.collocations import *
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
```

```
import operator
import re
from math import log10
import numpy


#general statistics that we'll calculate further down
numberofdocuments = 0
totalwordsperdocument = dict()
totalwordsincorpus = 0
totaltermsincorpus = 0

#maximum number of words in an N-gram (approximation)
maxN = 3


#CUSTOM TOKENIZER
#this tokenizer will split words, delete stop words and punctuation and
#transform it into uni/bi/trigrams, which will all be filtered by the regex.
#some document and general statistics are also calculated here.
punct = string.punctuation
punct = punct.translate(None,"'")
punctexcludeset = set(punct)
stop = set(stopwords.words('english'))
docindex = 0
def test_set(s):
    return ''.join(ch for ch in s if ch not in punctexcludeset)

def my_tokenizer(documentasstring):
    #these variables have to be declared here as global so the vectorizer can
    #see them from our program (we first declared them earlier)
    global docindex
    global totalwordsincorpus
    global totaltermsincorpus
    global numberofdocuments
    global maxN
    docsentences = nltk.sent_tokenize(documentasstring)
    docwords = list()
    doclength = 0
    for docsentence in docsentences:
        sentencenopunct = test_set(docsentence.lower())
        sentencewords = nltk.word_tokenize(sentencenopunct)
        doclength += len(sentencewords)
        for i in range(len(sentencewords)):
            word = sentencewords[i]
            if word[0] == "'":
                word = word[1:]
            sentencewords[i] = word
        sentenceclean = [i for i in sentencewords if i not in stop]
        docwords += sentenceclean
    totalwordsperdocument[docindex] = doclength
```

```
    totalwordsincorpus += doclength
    #all the words are split
    docbigrams = []
    doctrigrams = []
    for iword in range(len(docwords)):
        if maxN >= 2:
            if iword < len(docwords)-1:
                bigram = docwords[iword] + " " + docwords[iword+1]
                #print bigram
                docbigrams += [bigram]
                if maxN >= 3:
                    #print "enter max 3"
                    if iword < len(docwords)-2:
                        trigram = docwords[iword] + " " + docwords[iword+1] + " " +
docwords[iword+2]
                        doctrigrams += [trigram]
    docterms = docwords + docbigrams + doctrigrams
    totaltermsincorpus += len(docterms)
    docindex +=1
    numberofdocuments +=1
    return docterms


#PROCESSING:

#documents from FAO30 (same as exercise-2)

import os
path = "fao30/documents/"

all_docs = []
docindexnames = dict()
docreadindex = 0
for filename in os.listdir(path):
    docindexnames[docreadindex] = filename
    etd = open(path + filename)
    etdread = etd.read()
    etdread = etdread.decode('latin-1')
    etd_words = nltk.word_tokenize(etdread)
    all_docs += [etdread]
    docreadindex += 1


#uses a vectorizer to calculate term frequency
countVectorizer = CountVectorizer(tokenizer=my_tokenizer)
countVectorizer.build_analyzer()
#docstf = countVectorizer.fit_transform(set(["Alice stopped by the big big station
to retrieve the blue poop","Alice stopped by a poop and was angry"]))
docstf = countVectorizer.fit_transform(all_docs)
vecvocab = countVectorizer.vocabulary_
```

```
accumulatefreqDict = dict()
for termi in range(len(vecvocab)):
    acumforterm = docstf.getcol(termi).sum(0)[0]
    accumulatefreqDict[termi] = acumforterm


#calculates the phraseness for a term,document
def phraseness(documentn, termi):
    #pw = lmn bg - we use the approximation
    #qw = lm1 bg (unigram) - words are independent so we
    #just multiply their probabilities
    pw = float(accumulatefreqDict[termi])/float(totaltermsincorpus)
    termunigrams = termi.split()
    qw = 1
    for uni in termunigrams:
        qw *= float(accumulatefreqDict[uni])/float(totalwordsincorpus)
    return pw*log10(pw/qw)


#calculates the informativeness for a term,document
def informativeness(documentn, termi):
    #pw = lm1 fg (unigram)
    #qw = lm1 bg (unigram)
    # words are independent so we just multiply their probabilities
    termunigrams = termi.split()
    pw = 1
    for uni in termunigrams:
            pw *= float(docstf[documentn,
uni])/float(totalwordsperdocument[documentn])
    qw = 1
    for uni in termunigrams:
        qw *= float(accumulatefreqDict[uni])/float(totalwordsincorpus)
    return pw*log10(pw/qw)


#all the scores will be calculated here
dictscores = dict()
for doci in range(numberofdocuments):
    documentscores = dict()
    for termi in range(len(vecvocab)):
        documentscores[termi] = informativeness(doci, termi) + phraseness(doci,
termi)
    dictscores[doci] = documentscores


doccandidateslist = dict()
featurenames = list(countVectorizer.get_feature_names())
featurenamescopy = numpy.array(featurenames)
for idoc in range(numberofdocuments):
    probdoccopy = numpy.array(dictscores[idoc].values())
```

```
    #the keys were inserted by order, so the values were by this order as well
    sortedindices = (probdoccopy.argsort()[-5:])[::-1]
    candidatewordsfordoc = list()
    for candidatei in sortedindices:
        candidatewordsfordoc += [featurenamescopy[candidatei]]
    doccandidateslist[idoc] = candidatewordsfordoc

print doccandidateslist
```