Oscar Galindo & Aaron Alarcon

Dr. Nigel Ward

Topics in Language Processing

October 7, 2020

Assignment E

Introduction:

For this assignment we were tasked with implementing 4 different similarity algorithms that predict possible professors that are experts on some topics that are queried.

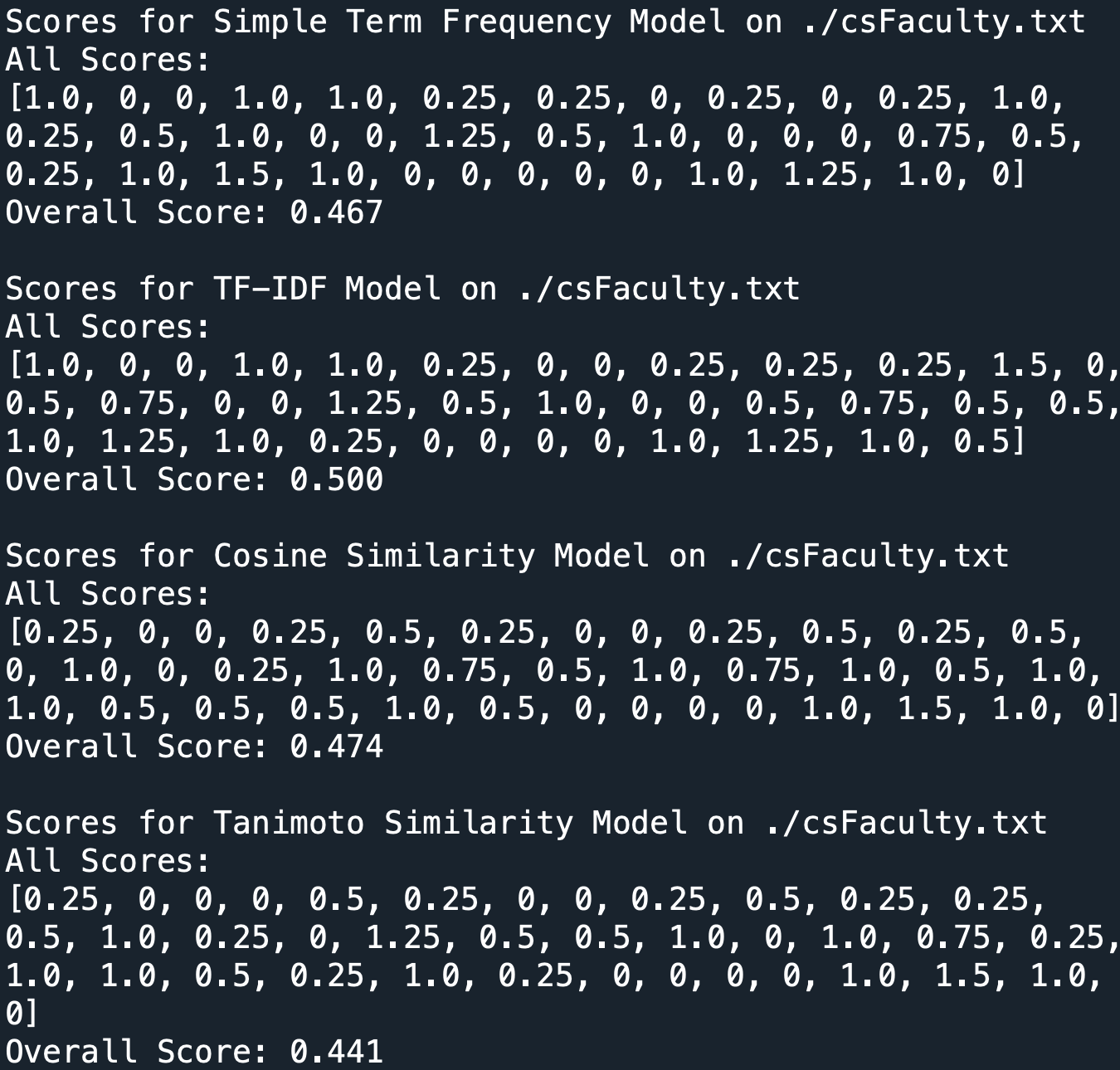
Design and Implementation:

For this exercise we first designed a naive approach for the similarities ranking based on term count. Such algorithm produce as a score the number of times that every word query appears in the target/evaluated document. The logic behind this idea is that if someone mentions once or twice a word of the query then the document describing the document being analyzed must certainly be dominated by other topics most likely those that are different than the queries’ topics. Otherwise, if a document makes the repeated use of the words of the query then the document/description must certainly be of someone who is an expert in the query.

We also decided to implement the TF-IDF model, and this model, as it is known, basically classifies the documents by how unique the words in document are. For example, if a term is mentioned three times among 1000 documents but all those mentions occur in the same document the certainly the word maps to a single document. This way we essentially try to understand how unique every word is and assess if for a term of the query a potential document target is defined by such word, if it is we choose it as a positive target, else we ignore it.

In addition, we decided to implement cosine similarity score as a measure of the similarities between a query and each of the description document. Naturally, due to the size of every description-document our similarities between the documents and the queries is very small, so just to look nice we normalize the probability mass between all the possible documents to obtain scores between 0-1 (this is not needed but made debugging faster). Also, we added stop words or a stop list which prevents considering the cases for unimportant words like “a”, “of”, etc.

Finally, we decided to implement tanimoto similarity score which follows the following equation: intersection of unique elements between query and document/ |number of unique elements in q| + |unique elements in document| - intersection of unique elements between query and document, which essentially evaluates what is the fraction of unique terms that is found in both documents in relation to the combined number of unique terms between both documents. This makes sense because we will like to observe the relation between both documents by analyzing how much both documents “refer to the same terms”.

Experimental Results:

Conclusions:

* Based on the previous results we observe that there is a strong correlation between the counts of terms and the targets that are ground truths, hence we suggest that our first hypothesis was correct.
* We partially think the results of the cosine similarity algorithm are correct because we have low similarities for each of queries when compared to the description files. This is consistent since we have very large description-documents and small query documents.
* We assume the Tanimoto similarity index is affected by the fact that professors tend to include descriptions that include words that are not really what their research is about but rather of outlying topics that connect to their main area of expertise. This at least to us explains the difference in scores between the cosine similarity score and the Tanimoto similarity index.
* We deduce that the reason why TF-IDF does very good with this data is because each of the queries contains a topic for which we assume that the best target index has at least most of the mentions when compared to the rest of the files. This way we conclude that there must be a decent amount of mentions of the topics that professor is an expert in, and she or he must be the person or one of the people with the most use of those words, which makes sense.

Appendix (Python 3.8):

# -\*- coding: utf-8 -\*-

"""

Created on Wed Oct 7 18:37:12 2020

@author: Aaron

"""

# Nigel Ward, UTEP, October 2018

# Updated by Angel Garcia, UTEP, July 2020

# Speech and Language Processing

# Assignment E: Information Retrieval

# This is just a skeleton that needs to be fleshed out.

# It is not intended as an example of good Python style

import numpy as np

import sys

import re

import math

def parseAlternatingLinesFile(file):

""" read a sequence of pairs of lines

e.g. text of webpage(s), name/URL

"""

sequenceA = []

sequenceB = []

with open(file, mode="r", encoding="utf-8") as f:

for i,line in enumerate(f):

if i % 2:

sequenceB.append(line.strip())

else:

sequenceA.append(line.strip())

return sequenceA, sequenceB

def generateCharTrigrams(text):

"""Generate Character Trigrams from Text"""

for i in range(len(text)-3+1):

yield text[i:i+3]

def computeFeatures(text, trigramInventory):

"""Computes the count of trigrams.

Trigrams can catch some similarities

(e.g. between "social" and "societal" etc.)

But really should be replaced with something better

"""

counts = {}

for trigram in generateCharTrigrams(text):

if trigram in trigramInventory:

counts[trigram] = (1 if trigram not in counts

else counts[trigram] + 1)

return counts

def computeSimilarity(dict1, dict2):

"""Compute the similarity between 2 dictionaries of trigtrams

Ad-hoc and inefficient.

"""

keys\_d1 = set(dict1.keys())

keys\_d2 = set(dict2.keys())

matches = keys\_d1 & keys\_d2

similarity = len(matches) / len(dict2)

#print(f"Similarity: {similarity:.3f}")

return similarity

def retrieve(queries, trigramInventory, archive):

"""returns an array: for each query, the top 3 results found"""

top3sets = []

for query in queries:

#print(f"query is {query}")

q = computeFeatures(query, trigramInventory)

#print(f"query features are \n{q}")

similarities = [computeSimilarity(q, d) for d in archive]

#print(similarities)

top3indices = np.argsort(similarities)[0:3]

#print(f"top three indices are {top3indices}")

top3sets.append(top3indices)

return top3sets

def valueOfSuggestion(result, position, targets):

weight = [1.0, .5, .25]

if result in targets:

return weight[max(position, targets.index(result))]

else:

return 0

def scoreResults(results, targets): #-----------------------------

merits = [valueOfSuggestion(results[i], i, targets)

for i in range(3)]

return sum(merits)

def scoreAllResults(queries, results, targets, descriptor,verbose=False):

print()

print(f"Scores for {descriptor}")

scores = [(q, r, t, scoreResults(r, t))

for q, r, t in zip(queries, results, targets)]

if verbose:

for q, r, t, s in scores:

print(f"for query: {q}")

print(f" results = \n{r}")

print(f" targets = \n{t}")

print(f" score = {s:.3f}")

all\_scores = [s for \_,\_,\_,s in scores]

overallScore = np.mean(all\_scores)

print(f"All Scores:\n{all\_scores}")

print(f"Overall Score: {overallScore:.3f}")

return overallScore

def pruneUniqueNgrams(ngrams):

twoOrMore = {}

print("Before pruning: " +

f"{len(ngrams)} ngrams across all documents")

twoOrMore = {k:v for k,v in ngrams.items() if ngrams[k] > 1}

print("After pruning: " +

f"{len(twoOrMore)} ngrams across all documents")

return twoOrMore

def targetNumbers(targets, nameInventory):

"""targets is a list of strings, each a sequence of names"""

targetIDs = []

for target in targets:

threeNumbers = []

for name in target.split():

threeNumbers.append(nameInventory.index(name))

targetIDs.append(threeNumbers)

return targetIDs

######################## Added Code ###########################

######################## TF-IDF ######################

def findWordsByDoc(doc):

return re.findall(r'"[a-z|A-Z]+[\'&-]\*[a-z|A-Z]\*"', doc)

def findWords(contents):

expression = re.compile("[a-z|A-Z]+\'\*[a-z|A-Z]\*")

words = {}

for document in range(len(contents)):

w = re.findall(expression, contents[document])

for word in w:

if not word in words:

words[word] = [0]\*len(contents)

for word in w:

words[word][document] += 1

return words

def findUniqueWords(words):

unique\_words = []

for w in words:

if not w in unique\_words:

unique\_words.append(w)

return unique\_words

def findWordCountsByDoc(contents):

expression = re.compile("[a-z|A-Z]+\'\*[a-z|A-Z]\*")

words = {}

for document in range(len(contents)):

w = re.findall(expression, contents[document])

unique = findUniqueWords(w)

for word in unique:

if not word in words:

words[word] = [0]\*len(contents)

for word in unique:

words[word][document] += 1

return words

def findAllNgrams(contents):

allTrigrams = {}

for text in contents:

for tri in generateCharTrigrams(text):

allTrigrams[tri] = (1 if tri not in allTrigrams

else allTrigrams[tri] + 1)

return allTrigrams

def retrieveTFIDF(queries,word\_frequency,term\_document\_frequency,archive):

top3sets = []

for query in queries:

q = findWordsByDoc(query)

similarities = [similaritiesTDIDF(len(archive),q,word\_frequency,term\_document\_frequency,archive[doc],doc)

for doc in range(len(archive))]

top3indices = np.argsort(similarities)[::-1][:3]

top3sets.append(top3indices)

return top3sets

def similaritiesTDIDF(archive\_size,query,word\_frequency,term\_document\_frequency,doc,doc\_index):

value = 0

for word in query:

if word in doc:

query\_weight = math.log(1 + (archive\_size/sum(term\_document\_frequency[word])))

term\_weight = 1 + math.log(word\_frequency[word][doc\_index])

value += term\_weight \* query\_weight

return value

######################## Cosine Similarity ######################

def cosine\_similarity(query,doc):

numerator = 0

denominator = 0

for key in query.keys():

if key in doc:

numerator += query[key]\*doc[key]

s = 0

for key in query.keys():

s+=query[key]\*\*2

denominator = math.sqrt(s)

s = 0

for key in doc.keys():

s+=doc[key]\*\*2

denominator \*= s

return (numerator/denominator)

def getSimilarities(query,contents, option):

s = []

for doc in contents:

if option == "Cosine":

r = cosine\_similarity(query, doc)

else:

r = Tanimoto\_sim(query, doc)

s.append(r)

normalize(s)

return s

def getWordsOfDocument(doc):

words = findWordsByDoc(doc)

w = dict()

for word in words:

if not word in w:

w[word] = 1

else:

w[word] += 1

return w

def getWordsOfDocumentSW(doc):

words = findWordsByDoc(doc)

w = dict()

for word in words:

if not inStopWords(word):

if not word in w:

w[word] = 1

else:

w[word] += 1

return w

def getWordsOfDocInCollection(contents,stop\_words=False):

docs =[]

for doc in contents:

if not stop\_words:

docs.append(getWordsOfDocument(doc))

else:

docs.append(getWordsOfDocumentSW(doc))

return docs

def normalize(vector):

s = sum(vector)

if s > 0:

for i in range(len(vector)):

vector[i] = vector[i]/s

def retrieveCosSim(queries,docs,stop\_words = False, option = "Cosine"):

top3sets = []

docs\_words = getWordsOfDocInCollection(docs)

for q in queries:

if not stop\_words:

q\_words = getWordsOfDocument(q)

else:

q\_words = getWordsOfDocumentSW(q)

similarities=getSimilarities(q\_words,docs\_words, option)

top3indices = np.argsort(similarities)[::-1][:3]

top3sets.append(top3indices)

return top3sets

######################## Stop List ######################

def inStopWords(word):

stop\_words,\_ = parseAlternatingLinesFile('./stopwords.txt')

#List taken from: https://www.lextek.com/manuals/onix/stopwords1.html

return word in stop\_words

########################

#Tanimoto Similarity

def Tanimoto\_sim(query, doc):

in\_common = 0

not\_in\_common = 0

for key in query.keys():

if key in doc:

not\_in\_common += abs(query[key] - doc[key])

in\_common += min(query[key], doc[key])

else:

not\_in\_common += query[key]

for label, value in doc.items():

if label not in query.keys():

not\_in\_common += value

if not\_in\_common == in\_common:

not\_in\_common +=1

return (in\_common/(not\_in\_common - in\_common))

######## TF #########

def termFrequenciesScore(query,contents):

scores = [0]\*len(contents)

q = getWordsOfDocument(query)

for doc in range(len(contents)):

words = getWordsOfDocument(contents[doc])

for term in q.keys():

if term in words:

scores[doc] += words[term]

return scores

def retrieveTF(queries,contents):

top3sets = []

for q in queries:

similarities = termFrequenciesScore(q, contents)

top3indices = np.argsort(similarities)[::-1][:3]

top3sets.append(top3indices)

return top3sets

##########################################################

if \_\_name\_\_ == "\_\_main\_\_":

# if len(sys.argv) != 3:

# print("Usage: python irStub.py " +

# "<document file>" +

# "<queries file>")

# sys.exit()

document\_file = "./csFaculty.txt"

queries = './trainingQueries.txt'

sys.argv = ['']\*3

sys.argv[1] = document\_file

sys.argv[2] = queries

print("......... irStub .........")

contents, names = parseAlternatingLinesFile(sys.argv[1])

print(f"read in pages for {names}")

# #################################

# trigramInventory = pruneUniqueNgrams(findAllNgrams(contents))

# archive = [computeFeatures(line, trigramInventory)

# for line in contents]

# queries, targets = parseAlternatingLinesFile(sys.argv[2])

# targetIDs = targetNumbers(targets, names)

# results = retrieve(queries, trigramInventory, archive)

# modelName = "silly character trigram model"

# scoreAllResults(queries, results, targetIDs,

# f"{modelName} on {sys.argv[1]}")

############# TF Similarities #############

queries, targets = parseAlternatingLinesFile(sys.argv[2])

print(termFrequenciesScore(queries[0],contents))

targetIDs = targetNumbers(targets, names)

results = retrieveTF(queries, contents)

modelName = "Simple Term Frequency Model"

scoreAllResults(queries, results, targetIDs,

f"{modelName} on {sys.argv[1]}")

############ TF-IDF #############

word\_frequency = findWords(contents)

term\_document\_frequency = findWordCountsByDoc(contents)

word\_frequencies = findAllNgrams(contents)

archive = [findUniqueWords(findWordsByDoc(line))

for line in contents]

queries, targets = parseAlternatingLinesFile(sys.argv[2])

targetIDs = targetNumbers(targets, names)

results = retrieveTFIDF(queries,word\_frequency,term\_document\_frequency,archive)

modelName = "TF-IDF Model"

scoreAllResults(queries, results, targetIDs,

f"{modelName} on {sys.argv[1]}")

########## Cosine Similarity ###########

docs\_words = getWordsOfDocInCollection(contents)

queries, targets = parseAlternatingLinesFile(sys.argv[2])

results = retrieveCosSim(queries,contents,False) #Change to True to Use Stop List

targetIDs = targetNumbers(targets, names)

modelName = "Cosine Similarity Model"

scoreAllResults(queries, results, targetIDs,

f"{modelName} on {sys.argv[1]}")

######### Tanimoto Similarity##########

results = retrieveCosSim(queries,contents,True,option = "Tanimoto") #Change to True to Use Stop List

targetIDs = targetNumbers(targets, names)

modelName = "Tanimoto Similarity Model"

scoreAllResults(queries, results, targetIDs,

f"{modelName} on {sys.argv[1]}")