CSE 5243 Lab 2 – Classification of Reuters Article Text

Afnan Rehman, Matthew Stock, & Kumar Dhital

Section 26469

November 6, 2018

# Feature Vector Design:

Our feature vectors were based off of TF-IDF models and a bag of words model. We felt these two vector types would provide a solid baseline classification system. We also thought it would be interesting to compare the two and see how accurate the different models would be in a real-life scenario.

We decided to create two vectors of each type, one for PLACES and one for TOPICS. The classes were sorted by individual items such as a specific place or topic, even if there were two or more grouped together for an article. The way we went about reconciling this is by providing a count for the word to each vector if two or more vectors intersected on an article.

For example, if an article contained a PLACES key for both ‘usa’ and ‘uk’, the body text would be counted towards both the vector for ‘usa’ and the vector for ‘uk’. This way we did not lose any information, and the door is left open in case you wanted to consider ‘usa’+’uk’ as a separate class altogether.

With this approach, we now had a full set of training data for our classifier, and we can also hold back some for the validation process by modifying which files are counted in the data set.

# Selected Classifier:

For this particular case, we chose a standard Naïve Bayes classifier using the Python NLTK language processing library to tokenize words. The approach gave us some interesting classification results while allowing us to compare our bag-of-words vector to our TF-IDF version.

# Training and Validation:

For this step, we used cross-validation as a way to segment and then validate the training data. We decided to split the data based on the file it was in, making the process of segmenting easy. Once we completed this, we averaged the results and compared.

# Results:

The results were expectedly not incredibly accurate but telling in several ways.

For our PLACES we were able to test validation data against training data and get consistent matches, as well as infinitesimally small numbers for probabilities against other countries, sometimes in the valley of 4.6 x 10-2000! Sorting these numbers in Excel (as we have them output to CSV) helped us understand which countries were more distinct from each other and which ended up matching very closely. For example, in our tests, we found Canada and the US to be extremely similar when we used our bag of words vector, causing the classifier to almost rank them equal in terms of probability against the test data.

# Appendix:

## Examples of Feature Vectors:

Bag of Words Vector (PLACE/TOPIC specific):

|  |  |
| --- | --- |
| ‘argentina’ | ‘Argentine grain board figures show crop registrations…’ |
| ‘bangladesh’ | ‘Bangladesh police mounted a cross-country…’ |
| ‘usa’ | ‘Showers continued throughout the week…’ |

OR

|  |  |
| --- | --- |
| ‘wheat’ | ‘…’ |
| ‘earn’ | ‘…’ |
| ‘retail’ | ‘…’ |

TF-IDF Vector:

## Source code for countTags.py

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

"""

@author: Afnan

Note: some of this is carried over from lab 1

"""

from bs4 import BeautifulSoup

from typing import Tuple

import itertools

import csv

# BEGIN PART FROM https://towardsdatascience.com/implementing-a-trie-data-structure-in-python-in-less-than-100-lines-of-code-a877ea23c1a1

class TrieNode(object):

"""

Trie node implementation.

"""

def \_\_init\_\_(self, char: str):

self.char = char

self.children = []

# Is it the last character of the word.

self.word\_finished = False

# How many times this character appeared in the addition process

self.counter = 1

def add(root, word: str):

"""

Adding a word in the trie structure

"""

node = root

for char in word:

found\_in\_child = False

# Search for the character in the children of the present `node`

for child in node.children:

if child.char == char:

# We found it, increase the counter by 1 to keep track that another

# word has it as well

child.counter += 1

# And point the node to the child that contains this char

node = child

found\_in\_child = True

break

# We did not find it so add a new chlid

if not found\_in\_child:

new\_node = TrieNode(char)

node.children.append(new\_node)

# And then point node to the new child

node = new\_node

# Everything finished. Mark it as the end of a word.

node.word\_finished = True

def find\_prefix(root, prefix: str) -> Tuple[bool, int]:

"""

Check and return

1. If the prefix exsists in any of the words we added so far

2. If yes then how may words actually have the prefix

"""

node = root

# If the root node has no children, then return False.

# Because it means we are trying to search in an empty trie

if not root.children:

return False, 0

for char in prefix:

char\_not\_found = True

# Search through all the children of the present `node`

for child in node.children:

if child.char == char:

# We found the char existing in the child.

char\_not\_found = False

# Assign node as the child containing the char and break

node = child

break

# Return False anyway when we did not find a char.

if char\_not\_found:

return False, 0

# Well, we are here means we have found the prefix. Return true to indicate that

# And also the counter of the last node. This indicates how many words have this

# prefix

return True, node.counter

# END PART FROM https://towardsdatascience.com/implementing-a-trie-data-structure-in-python-in-less-than-100-lines-of-code-a877ea23c1a1

# This method works like str.split, but splits for as many times as a delimiter shows up in the doc

# It is also original work based on prior knowledge of how string splits work in Python.

def multi\_splitter(input\_string, delimiter):

out\_strings = []

new\_sub = str(input\_string).split(delimiter)

for str\_element in new\_sub:

sub = str\_element.split("</D>")

out\_strings.append(sub[0])

return out\_strings

def get\_text(place, sources, places\_bag\_vector):

# This portion involving reading the body text in from the file mostly done by Kumar

print (place)

total\_text = ""

for source in sources:

print (source)

with open(source) as f:

data = f.read()

soup = BeautifulSoup(data, 'html.parser') # parse using HTML parser, close to structure of these files

reuters\_tags = soup.find\_all('reuters')

for reuter\_tag in reuters\_tags: # get information stored within each reuters tag

places\_tag = reuter\_tag.places

d\_tags = places\_tag.find\_all('d') # find all places/topics mentioned

for d\_tag in d\_tags:

for child in d\_tag.children: # find relevant tags to current call and add text to a master string

if(place == child):

try:

total\_text += reuter\_tag.body.get\_text()

except:

total\_text += ""

# This subsequent section is devoted to removing a few bits of rather unwieldy extra characters in our

# output string. We wanted to retain as many words as possible, so more tedious methods of extraction,

# such as removing '\n' from the MIDDLE of the word was required. This part written by Afnan.

array = total\_text.split()

new\_array = []

for word in array: # each word gets examined and picked apart if it contains the offending characters

new\_word = ""

if '\n' in word: # removing line breaks, wherever they may occur

subword = word.split('\n')

for part in subword:

if '\n' not in part:

new\_word += part

word = new\_word

new\_word = ""

if '.' in word: # removing punctuation

subword = word.split('.')

for part in subword:

if '.' not in part:

new\_word += part

word = new\_word

new\_word = ""

if ',' in word: # removing punctuation

subword = word.split(',')

for part in subword:

if ',' not in part:

new\_word += part

word = new\_word

new\_word = ""

if '"' in word: # removing punctuation

subword = word.split('"')

for part in subword:

if '"' not in part:

new\_word += part

word = new\_word

word += " "

new\_array.append(word)

cleaned\_text = ""

for newword in new\_array:# now removing some final pesky words as well as any numbers we don't want in our analysis

if "reuter" not in newword.lower() and "\x03" not in newword and '"' not in newword and newword.isdigit() == False:

cleaned\_text += newword

# Optionally, add the finished bag of words to a output file

cleaned\_text.rstrip()

file= open(place+'.txt', "a")

try:

file.write(cleaned\_text)

except:

file.write("")

file.close();

# Create vector and return to calling function

places\_bag\_vector[place] = cleaned\_text

# output looks like: {'afghanistan' : 'Pakistan complained to the United Nations today that...', 'algeria' : 'Liquefied natural gas imports from Algeria...', ....}

return places\_bag\_vector

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

sources = ["files/reut2-000.sgm", "files/reut2-001.sgm", "files/reut2-002.sgm", \

"files/reut2-003.sgm", "files/reut2-004.sgm", "files/reut2-005.sgm", \

"files/reut2-006.sgm", "files/reut2-007.sgm", "files/reut2-008.sgm", \

"files/reut2-009.sgm", "files/reut2-010.sgm", "files/reut2-011.sgm", \

"files/reut2-012.sgm", "files/reut2-013.sgm", "files/reut2-014.sgm", \

"files/reut2-015.sgm", "files/reut2-016.sgm", "files/reut2-017.sgm", \

"files/reut2-018.sgm"]

total\_blank\_places = 0

total\_blank\_topics = 0

total\_countries = []

total\_topics = []

root = TrieNode('\*')

# Here, my algorithm for splitting the elements of the TOPICS and PLACES fields is my original work

for source in sources:

print(source)

with open(source) as f: # Open the file and read line by line to a list array

array = []

for line in f:

array.append(line)

# Since PLACES were contained within one line of code according to the data I saw, I assumed

# that any line with the PLACES tag would contain all of the location info for that article

places = []

for index in array: # Look at lines containing the "PLACES" tag and read those into a separate list

if "<PLACES>" in index:

places.append(index)

# Once I got the line, I split the string on the multiple "<D>" tags to extract the location

# information within

new\_places = []

for place in places:

new\_places.extend(multi\_splitter(place, "<D>")) # Using the helpful method above, I split on one or more <D> tags

new\_places = [x for x in new\_places if x not in ('', '/', '\n', 'PLACES', '/PLACES')]# I then removed instances of tag information or blank information from the overall list

# One trick I learned in coding Python for work is that by casting a list as a set,

# you can remove duplicates in one line of code since sets do not contain duplicates

distinct\_countries = set(new\_places)

total\_countries.extend(distinct\_countries)

# Here I refer back to the original list of lines with the PLACES tag to locate blanks

# Since blank PLACES fields all had the same structure, counting them was a simple string comparison.

count\_blanks = 0

for place in places:

if place == "<PLACES></PLACES>\n":

count\_blanks += 1

total\_blank\_places += count\_blanks

# Next I moved onto TOPICS, using many of the same methods

# that I used for PLACES to count and extract the information

topics = []

for index in array:

if "<TOPICS>" in index:

topics.append(index)

# Once again I used the same string split method to extract the contents of each field

tops = []

for topic in topics:

tops.extend(multi\_splitter(topic, "<D>"))

tops = [x for x in tops if x not in ('', '/', '\n', 'TOPICS', '/TOPICS')]

# Counted distinct topics using the same cast to set

distinct\_topics = set(tops)

# You may notice the issue with simply extending the list of total topics

# There may end up being duplicates between documents that are not addressed

# I address this issue in the final step: printing the statistics after all loops are finished

total\_topics.extend(distinct\_topics)

count\_blanks = 0

for topic in topics:

if topic == "<TOPICS></TOPICS>\n":

count\_blanks += 1

total\_blank\_topics += count\_blanks

# Here we begin to make our country-based classifier

bag\_vector = {}

for country in sorted(set(total\_countries)): #this can take quite a while, leave commented out for performance

if "<PLACES>" not in country:

get\_text(country, sources, bag\_vector)

with open('bag\_train.csv', 'w') as csv\_file:

writer = csv.writer(csv\_file)

for key, value in bag\_vector.items():

writer.writerow([key, value])

## Source code for find\_trie\_prefix.py

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

"""

Created on Fri Oct 12 01:23:38 2018

@author: Afnan

"""

# BEGIN PART FROM https://towardsdatascience.com/implementing-a-trie-data-structure-in-python-in-less-than-100-lines-of-code-a877ea23c1a1

from typing import Tuple

class TrieNode(object):

"""

Trie node implementation.

"""

def \_\_init\_\_(self, char: str):

self.char = char

self.children = []

# Is it the last character of the word.

self.word\_finished = False

# How many times this character appeared in the addition process

self.counter = 1

def add(root, word: str):

"""

Adding a word in the trie structure

"""

node = root

for char in word:

found\_in\_child = False

# Search for the character in the children of the present `node`

for child in node.children:

if child.char == char:

# We found it, increase the counter by 1 to keep track that another

# word has it as well

child.counter += 1

# And point the node to the child that contains this char

node = child

found\_in\_child = True

break

# We did not find it so add a new chlid

if not found\_in\_child:

new\_node = TrieNode(char)

node.children.append(new\_node)

# And then point node to the new child

node = new\_node

# Everything finished. Mark it as the end of a word.

node.word\_finished = True

def find\_prefix(root, prefix: str) -> Tuple[bool, int]:

"""

Check and return

1. If the prefix exsists in any of the words we added so far

2. If yes then how may words actually have the prefix

"""

node = root

# If the root node has no children, then return False.

# Because it means we are trying to search in an empty trie

if not root.children:

return False, 0

for char in prefix:

char\_not\_found = True

# Search through all the children of the present `node`

for child in node.children:

if child.char == char:

# We found the char existing in the child.

char\_not\_found = False

# Assign node as the child containing the char and break

node = child

break

# Return False anyway when we did not find a char.

if char\_not\_found:

return False, 0

# Well, we are here means we have found the prefix. Return true to indicate that

# And also the counter of the last node. This indicates how many words have this

# prefix

return True, node.counter

# END PART FROM https://towardsdatascience.com/implementing-a-trie-data-structure-in-python-in-less-than-100-lines-of-code-a877ea23c1a1

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

root = TrieNode('\*')

# Here my algorithm for splitting off and counting the body words is my own

# Here the sources were hard coded. This can be easily changed to accept user input or to

# search through a list of files given a directory if needed

distinct\_words = []

sources = ["files/reut2-000.sgm", "files/reut2-001.sgm", "files/reut2-002.sgm", \

"files/reut2-003.sgm", "files/reut2-004.sgm", "files/reut2-005.sgm", \

"files/reut2-006.sgm", "files/reut2-007.sgm", "files/reut2-008.sgm", \

"files/reut2-009.sgm", "files/reut2-010.sgm", "files/reut2-011.sgm", \

"files/reut2-012.sgm", "files/reut2-013.sgm", "files/reut2-014.sgm", \

"files/reut2-015.sgm", "files/reut2-016.sgm", "files/reut2-017.sgm", \

"files/reut2-018.sgm", "files/reut2-019.sgm", "files/reut2-020.sgm", \

"files/reut2-021.sgm"]

for source in sources:

print("Parsing " + source[-13:])

with open(source) as f: # Open the file and read line by line into array

array = []

for line in f:

array.append(line)

words = []

body\_on = False

# Since there was no separator like the "<D>" used for TOPICS and PLACES,

# extracting the body text of the article only needed a standard strng split along one delimiter.

for index in array:

if "<BODY>" in index:

index = index.split("<BODY>",1)[1]

body\_on = True # flag is used to track when to start and stop adding lines to the body text

words.append(index)

if "</BODY>" in index:

index = index.split("</BODY>",1)[0]

body\_on = False

if body\_on == True:

words.append(index)

distinct\_words.extend(words)

# Words are then added to the prefix trie using the add function above in a loop

distinct\_words = set(distinct\_words) # list casted as set to get rid of duplicates

print("Adding words to prefix trie...")

for word in distinct\_words:

for word in distinct\_words:

try:

int(word) # here I try to filter out integers that would not be useful in the future.

except:

add(root,word)

print("Done!")

# counts of words can be found and printed using the find\_prefix methods as shown below

print(find\_prefix(root, 'limited'))

print(find\_prefix(root, 'the'))

## Source Code for getBodyTextBag.py

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

"""

@author: Kumar, Afnan

"""

from bs4 import BeautifulSoup # Make sure BeautifulSoup is installed on your device before running it

'''

This is the function takes PLACES as argument and takes body text from file for each body where PLACES happens.

It output whole body text to a file based on its <place>.txt name where place is the name of the country.

This method example could be placed in another script such as countTags.py to create vectors for PLACES and TOPICS separaately.

It counts only one country at a time, so if 'usa' is in an article along with 'uk', then the body of that article would

fall into both the USA key of the dictionary vector as well as the UK key of the vector.

'''

def get\_text(place, sources, places\_bag\_vector):

# This portion involving reading the body text in from the file mostly done by Kumar

total\_text = ""

for source in sources:

with open(source) as f:

data = f.read()

soup = BeautifulSoup(data, 'html.parser') # parse using HTML parser, close to structure of these files

reuters\_tags = soup.find\_all('reuters')

for reuter\_tag in reuters\_tags: # get information stored within each reuters tag

places\_tag = reuter\_tag.places

d\_tags = places\_tag.find\_all('d') # find all places/topics mentioned

for d\_tag in d\_tags:

for child in d\_tag.children: # find relevant tags to current call and add text to a master string

if(place == child):

total\_text += reuter\_tag.body.get\_text()

# This subsequent section is devoted to removing a few bits of rather unwieldy extra characters in our

# output string. We wanted to retain as many words as possible, so more tedious methods of extraction,

# such as removing '\n' from the MIDDLE of the word was required. This part written by Afnan.

array = total\_text.split()

new\_array = []

for word in array: # each word gets examined and picked apart if it contains the offending characters

new\_word = ""

if '\n' in word: # removing line breaks, wherever they may occur

subword = word.split('\n')

for part in subword:

if '\n' not in part:

new\_word += part

word = new\_word

new\_word = ""

if '.' in word: # removing punctuation

subword = word.split('.')

for part in subword:

if '.' not in part:

new\_word += part

word = new\_word

word += " "

new\_array.append(word)

cleaned\_text = ""

for newword in new\_array: # now removing some final pesky words as well as any numbers we don't want in our analysis

if "reuter" not in newword.lower() and "\x03" not in newword and newword.isdigit() == False:

cleaned\_text += newword

# Optionally, add the finished bag of words to a output file

file= open(place+'.txt', "a")

try:

file.write(cleaned\_text)

except:

file.write("")

file.close();

# Create vector and return to calling function

places\_bag\_vector[place] = cleaned\_text

return places\_bag\_vector

# output looks like: {'afghanistan' : 'Pakistan complained to the United Nations today that...', 'algeria' : 'Liquefied natural gas imports from Algeria...', ....}

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

sources = ["files/reut2-000.sgm", "files/reut2-001.sgm", "files/reut2-002.sgm", \

"files/reut2-003.sgm", "files/reut2-004.sgm", "files/reut2-005.sgm", \

"files/reut2-006.sgm", "files/reut2-007.sgm", "files/reut2-008.sgm", \

"files/reut2-009.sgm", "files/reut2-010.sgm", "files/reut2-011.sgm", \

"files/reut2-012.sgm", "files/reut2-013.sgm", "files/reut2-014.sgm", \

"files/reut2-015.sgm", "files/reut2-016.sgm", "files/reut2-017.sgm", \

"files/reut2-018.sgm", "files/reut2-019.sgm", "files/reut2-020.sgm", \

"files/reut2-021.sgm"]

vector = {}

vector = get\_text("nepal", sources, vector) # call method

print(vector)

# Source Code Citations:

Trie Source Code: <https://towardsdatascience.com/implementing-a-trie-data-structure-in-python-in-less-than-100-lines-of-code-a877ea23c1a1>