Popular Topic Mining from Blogs

Stone Fang (Student ID: 19049045)
Computers and Information Sciences
Auckland University of Technology
Auckland, New Zealand
fnk7060@autuni.ac.nz

I. Overview

People's concerns and opinions are important reference for innovations of new products or services. However, accomplishing such task by humans is expensive, time-consuming and difficult to scale. As a response, a number of individuals and organisations are leveraging text mining technologies to mining meaningful information from large volume of text such as news media [1]. Among a variety of studies and applications, topic modelling is an important method to extract hot topics which reflects public attention and opinion from massive texts [1]–[3]. However, effective method of extracting useful information from text on the Internet remains an open challenge [3].

Evaluation of topics mined from text is another challenge, mostly due to the lack of ground truth because topic modelling is an unsupervised learning task [4].

The goal of this project is to mine most popular topics that people were discussing from blog posts by utilising various text mining algorithms and tools. Specifically, we will find two most popular topics for each group in the following demographics:

- Males
- Females
- People younger than or equal to 20 years old
- People older than 20 years old
- Everyone

The remainder of this article is organised as follows. In section II related works on topic mining and evaluation will be reviewed. The methodology of topic mining are detailed in section III, while the results, analysis and evaluations are presented in section IV. The works of this article are summarised in section V and open issues and future works are discussed in section VI.

II. Related Work

Jacobi, Atteveldt, and Welbers [1] conducted an in-depth study of how to apply topic modelling technologies on analysis of qualitative data in academic research.

III. Research Design

In this section the solution will be described in detail. First an overview of the dataset is given, and then the algorithm of topic mining is detailed.

A. Data Description

The dataset contains 19,320 files in XML format, each containing articles of one person posted generally between 2001 and 2004. Metadata of the bloggers includes gender, age, category, and zodiac. In addition, the number of posts for each person are also counted. The result is summarised in figure 1. From this figure we can acquire some basic statistics of the dataset, including

1) Gender: data samples are quite evenly distributed over both genders.
2) Age: most bloggers are younger than 30, almost of them under 20. On the other hand, there are two gaps around 20 and 30 which may implies some missing data points in the dataset
3) Category: the most frequent category is unknown, which is trivial, while the second frequent one is student, far more than other categories.
4) Zodiac: The distribution over zodiac is reasonable even.
5) Number of posts: most bloggers published less than 100 posts, while the peak appears at 10, which implies people are most likely to write around 10 posts.

B. Topic Mining Algorithm

The general idea for mining popular topics used in this project is to find the most significant "things" mentioned in the overall dataset, as well as the closely related information.

The overall architecture of the algorithm is shown as figure 2.

1) Data Cleaning:
2) Tokenization:
3) POS Tagging:
4) NER:
5) Stopwords Removal:
6) Stemming and Lemmatization:
7) Counting and TF-IDF:

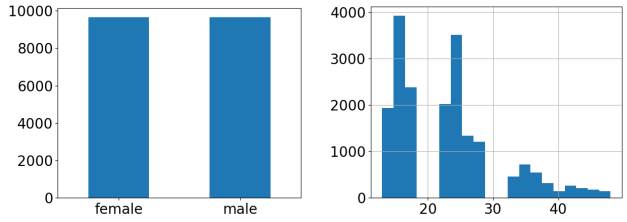
IV. Results, Analysis, and Evaluation

V. Conclusion

VI. Open Issues and Future Works

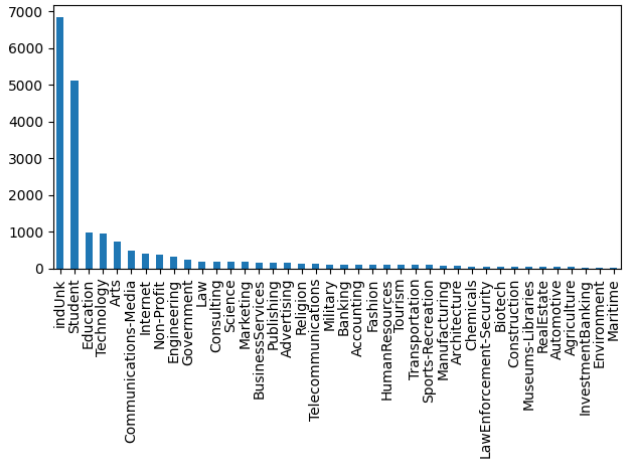
References

[1] C. Jacobi, W. v. Atteveldt, and K. Welbers, "Quantitative analysis of large amounts of journalistic texts using topic modelling," *Digital Journalism*, vol. 4, no. 1, pp. 89–106, Jan. 2, 2016, ISSN: 2167-0811. DOI: 10.1080/21670811.2015.1093271.

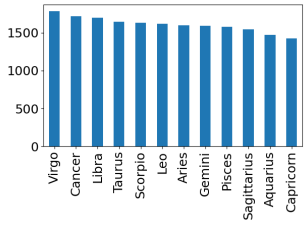


(a) Gender

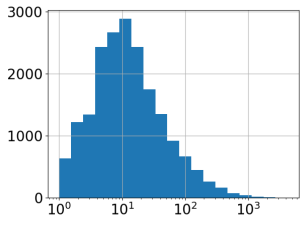
(b) Age



(c) Category



(d) Zodiac



(e) Number of Posts

Fig. 1. Data Overview. Histogram over (a) gender (b) age (c) category (d) zodiac (c) number of posts

[2] P. Waila, V. K. Singh, and M. K. Singh, "Blog text analysis using topic modeling, named entity recognition and sentiment classifier combine," in *2013 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Mysore: IEEE, Aug. 2013, pp. 1166–1171, ISBN: 978-1-4673-6217-7 978-1-4799-2432-5 978-1-4799-2659-6. DOI: 10.1109/ICACCI.2013.6637342.

[3] J. Guo, P. Zhang, JianlongTan, and L. Guo, "Mining hot topics from twitter streams," *Procedia Computer Science*, vol. 9, pp. 2008–2011, 2012, ISSN: 18770509. DOI: 10.1016/j.procs.2012.04.224.

[4] J. Boyd-Graber, D. Mimno, and D. Newman, "Care and feeding of topic models: Problems, diagnostics, and improvementes," *Handbook of Mixed Membership Models and Their Applications*, p. 30,

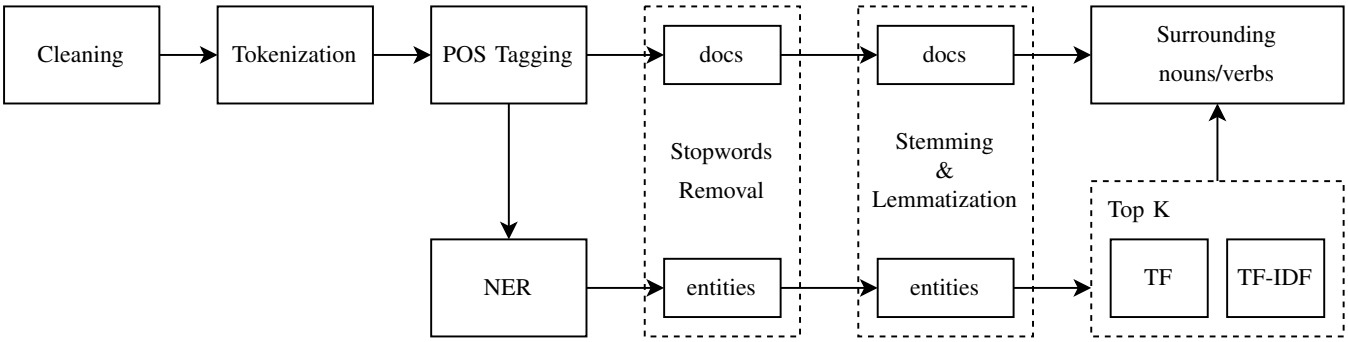


Fig. 2. Algorithm

APPENDIX
SOURCE CODE IN PYTHON

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import sys
import os.path
from glob import glob
from tqdm import tqdm
import pickle
import json
from datetime import date
import pprint
pp = pprint.PrettyPrinter(indent=2)

import random
import itertools
from collections import namedtuple, Counter, OrderedDict, defaultdict
import heapq
from operator import itemgetter
import re
from bs4 import BeautifulSoup
import numpy as np

from spellchecker import SpellChecker
import nltk
from nltk.corpus import stopwords
from nltk.corpus import wordnet
from nltk.stem import PorterStemmer, LancasterStemmer, WordNetLemmatizer

NUM_SAMPLES = None
_DEBUG = False

_DEBUG = True

STOPWORDS = set(stopwords.words("english"))
# Add more stopwords manually
with open('stopwords1.txt') as f:
    STOPWORDS.update(w.strip().lower() for w in f)
STOPWORDS.update(['i\'m', 'dont', '\'t', '\'m', '\'s', '\'re', '\'ve',
    'haha', 'hah', 'wow', 'hehe', 'heh',
    'ah', 'ahh', 'hm', 'hmm', 'urllink', 'ok', 'hey', 'yay', 'yeah'])

################################################################################
#                          Utility functions                                   #
################################################################################

def len2d(iter2d):
    return sum(len(d) for d in iter2d)

def list2d(iter2d):
    return [[x for x in inner] for inner in iter2d]

def flatten2d(list2d):
    return itertools.chain.from_iterable(list2d)

def flatten3d(list3d):
    return itertools.chain.from_iterable(flatten2d(list3d))

def mapbar(f, seq, desc):
    for e in tqdm(seq, desc):
        yield f(e)

def map2d(f, docs):
    with tqdm(total=len2d(docs)) as pbar:
        def _helper(sent):
```

```
            pbar.update(1)
            return f(sent)

        return [list(map(_helper, doc)) for doc in docs]

def map3d(f, docs):
    with tqdm(total=len2d(docs)) as pbar:
        def _helper(sent):
            pbar.update(1)
            return [f(word) for word in sent]

        return [list(map(_helper, doc)) for doc in docs]

def foreach3d(f, docs):
    with tqdm(total=len2d(docs)) as pbar:
        for doc in docs:
            for sent in doc:
                for word in sent:
                    f(word)
                pbar.update(1)

def foreach2d(f, docs):
    with tqdm(total=len2d(docs)) as pbar:
        for doc in docs:
            for sent in doc:
                f(sent)
                pbar.update(1)

def filter3d(f, docs):
    ret = []
    with tqdm(total=len2d(docs)) as pbar:
        def _helper_doc(doc):
            for sent in doc:
                pbar.update(1)
                out = [word for word in sent if f(word)]
                if len(out) > 0:
                    yield out

        for doc in docs:
            out = list(_helper_doc(doc))
            ret.append(out)
    return ret

def load_pkl(fpath):
    print('load dataset from cached pickle file ' + fpath)
    with open(fpath, 'rb') as f:
        dataset = pickle.load(f)
    return dataset

def save_pkl(obj, fpath):
    with open(fpath, 'wb') as f:
        print('save dataset to pickle file ' + fpath)
        pickle.dump(obj, f)

def save_json(obj, fpath, indent=2):
    with open(fpath, 'w', encoding="utf8") as f:
        print('save dataset to json file ' + fpath)
        json.dump(obj, f, indent=indent)

##############################################################################
#               Codes for data reading & transformation                      #
##############################################################################

Record = namedtuple('Record', ['meta', 'posts'])
Post = namedtuple('Post', ['date', 'text'])
MetaData = namedtuple('MetaData', ['id', 'gender', 'age', 'category', 'zodiac'])

```

```
def parse_meta_data(meta_data_str):
    arr = meta_data_str.lower().strip().split('.')
    return MetaData(arr[0], arr[1], int(arr[2]), arr[3], arr[4])

def read_blog_file(fpath):
    try:
        with open(fpath, encoding='utf-8', errors='ignore') as f:
            soup = BeautifulSoup(f.read(), "xml")
        blog = soup.Blog
    except ParseError:
        print('Error: invalid xml file {}'.format(fpath))
        raise
        return []

    posts = []
    state = 'date'
    for c in blog.find_all(recursive=False):
        if c.name != state:
            print('Warning: inconsistent format in file {}'.format(fpath))
        if state == 'date':
            try:
                date_str = c.text.strip()
                date = date_str
            except ValueError:
                print('Warning: invalid date {} in file {}' \
                        .format(c.text, fpath))
            state = 'post'
        else:
            text = c.text.strip()
            state = 'date'
            posts.append(Post(date, text))
    posts.sort(key=lambda p: p.date)
    return posts

def read_blogs(path, force=False, cache_file='blogs.pkl'):
    if not force and cache_file is not None and os.path.exists(cache_file):
        return load_pkl(cache_file)

    dataset = read_blogs_xml(path)

    # save to pickle file for fast loading next time
    if cache_file is not None:
        save_pkl(dataset, cache_file)

    return dataset

def read_blogs_xml(path):
    print('reading all data files from directory {} ...'.format(path))
    dataset = []

    if _DEBUG:  # use small files for fast debugging
        files = [os.path.join(path, fname) for fname in ['3998465.male.17.indUnk.Gemini.xml',
            '3949642.male.25.indUnk.Leo.xml', '3924311.male.27.HumanResources.Gemini.xml']]
        files = random.sample(list(glob(os.path.join(path, '*'))), 100)
    elif NUM_SAMPLES is None:
        files = glob(os.path.join(path, '*'))
    else:
        files = random.sample(list(glob(os.path.join(path, '*'))), NUM_SAMPLES)

    for fpath in  tqdm(files):
        fname = os.path.basename(fpath)
        meta_data = parse_meta_data(fname)
        posts = read_blog_file(fpath)
        rec = Record(meta_data, posts)
        dataset.append(rec)
    return dataset

```

```
199 ################################################################################
200 #              Codes for topic mining                                          #
201 ################################################################################
202
203 punct_re = re.compile(r'([\.!?,:;])(?=[a-zA-Z])')  # add space between a punctuation and a word
204 # replace two or more consecutive single quotes to a double quote
205 #   e.g. '' -> "       ''' -> "
206 quotes_re = re.compile(r"[\']{2,}")
207 def preprocess(text):
208     out = punct_re.sub(r'\1 ', text)
209     out = quotes_re.sub(r'"', out)
210     out = remove_invalid(out)
211     return out
212
213 leading_quote_re = re.compile(r'[\'\.~=*&^%#!|\-]+([a-zA-Z].*)')
214 def clean_word(word):
215     if word in ("'ve", "'re", "'s", "'t", "'ll", "'m", "'d", "'", "''"):
216         return word
217     word = leading_quote_re.sub(r'\1', word)
218     return word.strip()
219
220 def tokenise(dataset):
221     '''
222     consider all the blogs from one person as a document
223
224     Returns
225     ---------
226     docs: list of list of list
227         a list of documents, each of which is a list of sentences,
228         each of which is a list of words.
229     '''
230
231     print('tokenising the text dataset...')
232     docs = []
233     with tqdm(total=sum(len(rec.posts) for rec in dataset)) as pbar:
234         for rec in dataset:
235             doc = []
236             for post in rec.posts:
237                 for sent_str in nltk.sent_tokenize(post.text):
238                     sent_str = preprocess(sent_str)
239                     sent = [clean_word(w) for w in nltk.word_tokenize(sent_str)]
240                     sent = [w for w in sent if w != '']
241                     doc.append(sent)
242                 pbar.update(1)
243             docs.append(doc)
244
245     return docs
246
247 def calc_vocab(docs):
248     '''Calculate the vocabulary (set of distinct words) from a collection
249       of documents.
250     '''
251
252     print('calculating the vocabulary...')
253     vocab = set()
254
255     def _helper(sent):
256         vocab.update(sent)
257
258     foreach2d(_helper, docs)
259     return sorted(vocab)
260
261 def calc_pos_tags(docs):
262     print('POS tagging...')
263     def _f(sent):
264         try:
265             return nltk.pos_tag(sent)
```

```
        except IndexError:
            print('error sentence: {}'.format(sent))
            raise
    tagged_docs = map2d(_f, docs)
    return tagged_docs

pattern = re.compile(r'([^\.])\1{2,}')
pattern_ellipse = re.compile(r'\.{4,}')
invalid_chars = re.compile(r'[*\^#]')
def remove_invalid(text):
    '''Basic cleaning of words, including:

      1. rip off characters repeated more than twice as English words have a max
         of two repeated characters.
      2. remove characters which are not part of English words
    '''

    text = invalid_chars.sub(' ', text)
    text = pattern.sub(r'\1\1', text)
    text = pattern_ellipse.sub('...', text)
    return text.strip()

def remove_invalid_all(docs):
    print('reduce lengthily repreated characters...')
    return filter3d(lambda w: len(w) > 0, map3d(remove_invalid, docs))

spell = SpellChecker()

def correct_spelling(word):
    if not wordnet.synsets(word) and not word in STOPWORDS:
        return spell.correction(word)
    else:
        return word

def correct_spelling_all(docs):
    print('running spelling correction...')
    return map3d(correct_spelling, docs)

def remove_stopwords(docs):
    print('removing stopwords...')
    return filter3d(lambda wp: wp[0].lower() not in STOPWORDS, docs)

lemmatizer = WordNetLemmatizer()
porter = PorterStemmer()
lancaster = LancasterStemmer()
def stem_word(word):
    return porter.stem(lemmatizer.lemmatize(word))

def do_stemming(docs):
    print('stemming or lemmatising words...')
    return map3d(lambda wp: (stem_word(wp[0]), wp[1]), docs)

def calc_ne_all(docs):
    print('extracting named entities...')
    def _calc_ne(sent):
        ne = []
        for chunk in nltk.ne_chunk(sent):
            if hasattr(chunk, 'label'):
                ne.append((' '.join(c[0] for c in chunk), chunk.label()))
        return ne
    return map2d(_calc_ne, docs)


def calc_df(docs):
    df = defaultdict(lambda: 0)
    for doc in docs:
        for w in set(doc):
```

```
            df[w] += 1
    return df

def calc_tfidf(docs):
    '''The original TF-IDF is a document-wise score. This function will
    calculate the average TF-IDF on whole dataset as an overall scoring.
    '''
    tf_idf = defaultdict(lambda: 0)
    df = calc_df(docs)
    num_docs = len(docs)
    for doc in docs:
        counter = Counter(doc)
        num_words = len(doc)
        for token in set(doc):
            tf = counter[token] / num_words
            df_i = df[token]
            idf = np.log(num_docs / df_i)
            tf_idf[token] += tf * idf

    for token in tf_idf:
        tf_idf[token] /= num_docs

    return tf_idf

def get_top_topics(named_entities, n=5, method='tf'):
    print('calculating most popular topics by ' + method + '...')
    if method == 'tf':
        ranks = nltk.FreqDist(w for w, t in flatten3d(named_entities))
        print(ranks.most_common(50))
        ranks = dict(ranks)
    elif method == 'tfidf':
        ranks = calc_tfidf([[w for w, t in flatten2d(doc)] for doc in named_entities])
    ranks = [(k, v) for k, v in ranks.items()]
    print('n largest:', heapq.nlargest(200, ranks, key=itemgetter(1)))
    topics = heapq.nlargest(n, ranks, key=itemgetter(1))
    print('topics: ', topics)
    return topics

def get_surroundings(words, docs, n=4):
    '''expand the topic to be 2 verb/noun before and 2 verb/noun after the topic
    '''

    print('get surrounding 2 nouns/verbs for words {}'.format(words))

    sur = {}
    for w, c in words:
        sur[w] = Counter()

    # POS tags list for searching verbs/nouns

    def _helper(sent):
        sent_w = [w for w, p in sent]
        for w, c in words:
            try:
                idx = sent_w.index(w)
            except ValueError:
                continue

            after = 0
            vicinity = [sent[i] for i in [idx-2, idx-1, idx+1, idx+2]
                    if i >= 0 and i < len(sent)]
            for (wi, pi) in vicinity:
                if pi.startswith('N') or pi.startswith('V'):
                    sur[w][wi] += 1

    foreach2d(_helper, docs)
    ret = []
```

```
    for w, c in words:
        ret.append({'topic': w, 'score': c, 'keywords': sur[w].most_common(n)})
    return ret

def calc_intermediate_data(dataset):
    docs = tokenise(dataset)
    vocab = calc_vocab(docs)
    print('Size of vocabulary: {}'.format(len(vocab)))
    print(vocab[1:2000:2])
    print(vocab[1:100000:100])



    tagged_docs = calc_pos_tags(docs)
    docs = vocab = None

    named_entities = calc_ne_all(tagged_docs)

    # Remove stopwords after POS tagging and NER finished
    tagged_docs = remove_stopwords(tagged_docs)
    named_entities = remove_stopwords(named_entities)

    tagged_docs = do_stemming(tagged_docs)
    named_entities = do_stemming(named_entities)
    return tagged_docs, named_entities

def mine_topics(dataset, intermediate_data, group='all'):
    print('-' * 80)
    print('mining most popular topics for group ' + group)
    print('-' * 80)
    tagged_docs, named_entities = intermediate_data

    if group != 'all':
        if group == 'male' or group == 'female':
            idx = [i for i, rec in enumerate(dataset) if rec.meta.gender == group]
        elif group == '<=20':
            idx = [i for i, rec in enumerate(dataset) if rec.meta.age <= 20]
        elif group == '>20':
            idx = [i for i, rec in enumerate(dataset) if rec.meta.age > 20]
        else:
            raise NotImplementedError()
        tagged_docs = [tagged_docs[i] for i in idx]
        named_entities = [named_entities[i] for i in idx]

    print('selected docs: {}, {}'.format(len(tagged_docs), len(named_entities)))

    ret = {}
    print('-------------- result from TFIDF ------------------')
    topics = get_top_topics(named_entities, n=50, method='tfidf')
    keywords = get_surroundings(topics, tagged_docs, n=200)
    ret['tfidf'] = keywords

    print('-------------- result from TF ------------------')
    topics = get_top_topics(named_entities, n=50, method='tf')
    keywords = get_surroundings(topics, tagged_docs, n=20)
    ret['tf'] = keywords
    return ret

def main_intermediate():
    if not _DEBUG and NUM_SAMPLES is None:
        dataset = read_blogs('blogs')
    else:
        dataset = read_blogs('blogs', cache_file=None)

    intermediate_data = calc_intermediate_data(dataset)
    save_pkl(intermediate_data, 'intermediate_data.pkl')
    return dataset, intermediate_data
```

```
467
468 def main_mine_topics(dataset=None, intermediate_data=None):
469     if dataset is None:
470         dataset = load_pkl('blogs.pkl')
471     if intermediate_data is None:
472         intermediate_data = load_pkl('intermediate_data.pkl')
473
474     topics = {}
475     topics['male'] = mine_topics(dataset, intermediate_data, group='male')
476     topics['female'] = mine_topics(dataset, intermediate_data, group='female')
477     topics['no_more_than_20'] = mine_topics(dataset, intermediate_data, group='<=20')
478     topics['more_than_20'] = mine_topics(dataset, intermediate_data, group='>20')
479     topics['all'] = mine_topics(dataset, intermediate_data, group='all')
480     if _DEBUG:
481         suffix = 'debug'
482     else:
483         suffix = date.today().strftime('%Y%m%d')
484         if NUM_SAMPLES > 0:
485             suffix += '-' + str(NUM_SAMPLES)
486
487     save_json(topics, 'topics-{}.json'.format(suffix))
488
489 def main():
490     if len(sys.argv) <= 1:
491         phases = [1, 2]
492     else:
493         phases = [int(i) for i in sys.argv[1].split(',')]
494
495     dataset = intermediate_data = None
496     for ph in phases:
497         if ph == 1:
498             dataset, intermediate_data = main_intermediate()
499         elif ph == 2:
500             main_mine_topics(dataset, intermediate_data)
501
502 if __name__ == '__main__':
503     main()
```