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

A. Objective

II. Related Work

III. Research Design

A. Data Description

The dataset contains 19,320 files in XML format, each containing articles of one person posted generally between 2001 and 2004.

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 1.

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, Evaluation, and Accuracy Insurance

V. Conclusion, Open Issues and Future Work

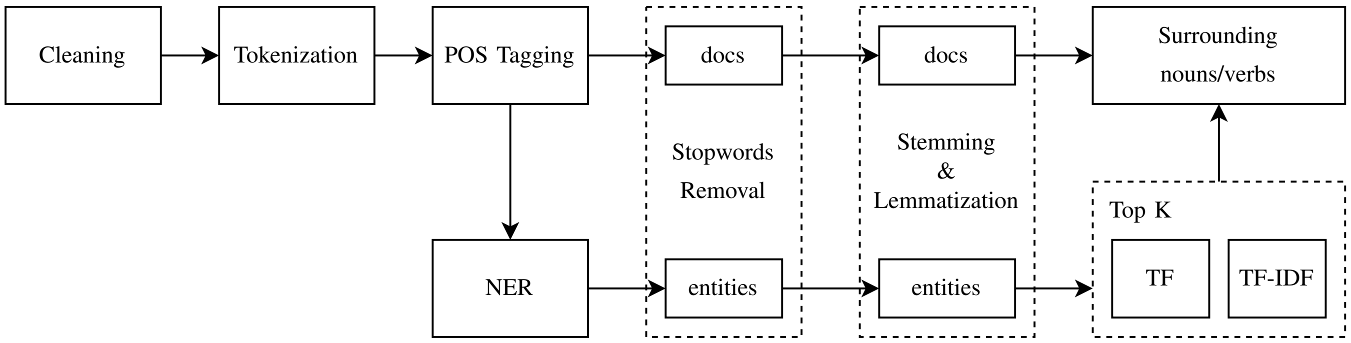

Fig. 1. Algorithm

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 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
import pandas as pd

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

_DEBUG = True

STOPWORDS = set(stopwords.words("english"))
# Add more stopwords manually
STOPWORDS.update(['i\'m', 'dont', '\'t', '\'m', '\'s', '\'re', '\'ve',
    'haha', 'hah', 'wow', 'hehe', 'heh',
    'ah', 'ahh', 'hm', 'hmm', 'urllink', 'ok', 'hey', 'yay', 'yeah'])
print('stop words:', STOPWORDS)

################################################################################
#                         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]

```

```python
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

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

    dataset = read_blogs_xml(path)

    # save to pickle file for fast loading next time
    if cache_file is not None:
        with open(cache_file, 'wb') as f:
            print('save dataset to pickle file ' + cache_file)
            pickle.dump(dataset, f)

    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']]
    else:
        files = glob(os.path.join(path, '*'))

    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

def show_summary(dataset):
    '''This function describes the summary of dataset or human inspection.
    It's not necessary for the mining process.

    Parameters
    --------------
    dataset : list of Record
        The blog dataset
    '''

    df = pd.DataFrame([d.meta for d in dataset])
    df['blog_count'] = [len(d.posts) for d in dataset]
    print(df.describe(include='all'))
    print('{} possible values for "gender": {}'.format(
            len(df.gender.unique()), ', '.join(sorted(df.gender.unique()))))
    print('{} possible values for category: {}'.format(
            len(df.category.unique()), ', '.join(sorted(df.category.unique()))))
    print('{} possible values for zodiac: {}'.format(
            len(df.zodiac.unique()), ', '.join(sorted(df.zodiac.unique()))))
```

```python

################################################################################
#               Codes for topic mining                                      #
################################################################################

punct_re = re.compile(r'([\.!?,:;])(?=[a-zA-Z])')  # add space between a punctuation and a word
# replace two or more consecutive single quotes to a double quote
#   e.g. '' -> "      ''' -> "
quotes_re = re.compile(r"[\']{2,}")
def preprocess(text):
    out = punct_re.sub(r'\1 ', text)
    out = quotes_re.sub(r'"', out)
    out = remove_invalid(out)
    return out

def tokenise(dataset):
    '''
    consider all the blogs from one person as a document

    Returns
    ---------
    docs: list of list of list
        a list of documents, each of which is a list of sentences,
        each of which is a list of words.
    '''

    print('tokenising the text dataset...')
    docs = []
    with tqdm(total=sum(len(rec.posts) for rec in dataset)) as pbar:
        for rec in dataset:
            doc = []
            for post in rec.posts:
                for sent_str in nltk.sent_tokenize(post.text):
                    sent_str = preprocess(sent_str)
                    sent = [w for w in nltk.word_tokenize(sent_str)]
                    doc.append(sent)
                pbar.update(1)
            docs.append(doc)

    return docs

def calc_vocab(docs):
    '''Calculate the vocabulary (set of distinct words) from a collection
      of documents.
    '''

    print('calculating the vocabulary...')
    vocab = set()

    def _helper(sent):
        vocab.update(sent)

    foreach2d(_helper, docs)
    return sorted(vocab)

def calc_pos_tags(docs):
    print('POS tagging...')
    def _f(sent):
        try:
            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(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 [w for (w, c) in topics]

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

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

    sur = OrderedDict()
    for w in words:
        sur[w] = Counter()

    # POS tags list for searching verbs/nouns
    target_pos_tags = ('NN', 'NNS', 'NNP', 'NNPS', 'VB', 'VBP', 'VBD', 'VBN',
            'VBG', 'VBZ')

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

            after = 0
            for (wi, pi) in sent[(idx+1):]:
                if pi in target_pos_tags:
                    sur[w][wi] += 1
                    after += 1
                if after == window:
                    break

            before = 0
            for (wi, pi) in reversed(sent[:idx]):
                if pi in target_pos_tags:
                    sur[w][wi] += 1
                    before += 1
                if before == window:
```

```python
                    break

    foreach2d(_helper, docs)
    ret = OrderedDict()
    for k, c in sur.items():
        ret[k] = c.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[:500])



    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)))

    print('-------------- result from TFIDF ------------------')
    topics = get_top_topics(named_entities, n=50, method='tfidf')
    print('most popular topics by TFIDF: {}'.format(topics))
    keywords = get_surroundings(topics, tagged_docs, n=20, window=2)
    pp.pprint(keywords)

    print('-------------- result from TF ------------------')
    topics = get_top_topics(named_entities, n=50, method='tf')
    print('most popular topics by TF: {}'.format(topics))
    keywords = get_surroundings(topics, tagged_docs, n=20, window=2)
    pp.pprint(keywords)


def main():
    if _DEBUG:
        dataset = read_blogs('blogs', cache_file=None)
    else:
        dataset = read_blogs('blogs')

    intermediate_data = calc_intermediate_data(dataset)
    mine_topics(dataset, intermediate_data, group='male')
    mine_topics(dataset, intermediate_data, group='female')
    mine_topics(dataset, intermediate_data, group='<=20')
    mine_topics(dataset, intermediate_data, group='>20')
    mine_topics(dataset, intermediate_data, group='all')
    return

if __name__ == '__main__':
    main()
```