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 20 years old or younger
- People older than 20
- Everyone

The remainder of this article is organised as follows. In **??** 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. Literature Review

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.

Boyd-Graber, Mimno, and Newman [4] provides a summary of topic evaluation methods, which are divided into three categories: human evaluation, diagnostic metrics, and coherent metrics. The first one needs human effort so it is expensive and time-consuming, while the other two can be calculated by computer without human interference.

Human Evaluation requires human involvement in the evaluation task. One method in this category is accomplished by word intrusion task. Specifically, a person will be presented by a list of words and is asked to find an intruder in the meaning of not belonging to others. The words list are constructed by first selecting highly possible words from a topic, and then randomly choose one word with low probability in the same topic but high probability in a different topic. If the intruders are easily to be identified, then the topic is more likely coherent [4].

Diagnostic Metrics only compute statistics of topics without requirements of external knowledge source. Some methods in this category are [4]:

- Topic Size: measured by the sum of numbers of tokens belonging to a certain topic. Generally speaking, small topic size means low quality.
- Word Length: average length of N most dominant words in a topic. The usefulness of this metric is corpus dependent.
- Corpus Distribution Distance: A probability distribution can be derived from a topic over the vocabulary, and further normalised by global word count in the whole dataset. The distance between different topics reflects how much these topics are separated.

Coherence Metrics is a type of methods which automatically compute score of topic coherence, and their accuracy is close to human performance. The basic idea is measuring how a pair of words from top N dominant words are associated [4]. It is formalised as

$$\text{TC-f}(\mathbf{w}) = \sum_{i<j} f(w_i, w_j), i, j \in \{1...N\}$$

where $\mathbf{w} = \{w_1, w_2, ..., w_N\}$ is the list of N most dominant words, and f is the scoring function of association between two words. There are a variety of ways to compute f, such as counting the co-occurrence of two words, or counting the number of documents containing both words. Two popular implementations of f are pointwise mutual information (PMI) and log conditional probability (LCP) [4].

$$\text{PMI}(w_i, w_j) = \log \frac{P(w_i, w_j)}{P(w_i)P(w_j)}$$
$$\text{LCP}(w_i, w_j) = \log \frac{P(w_i, w_j)}{P(w_j)}$$

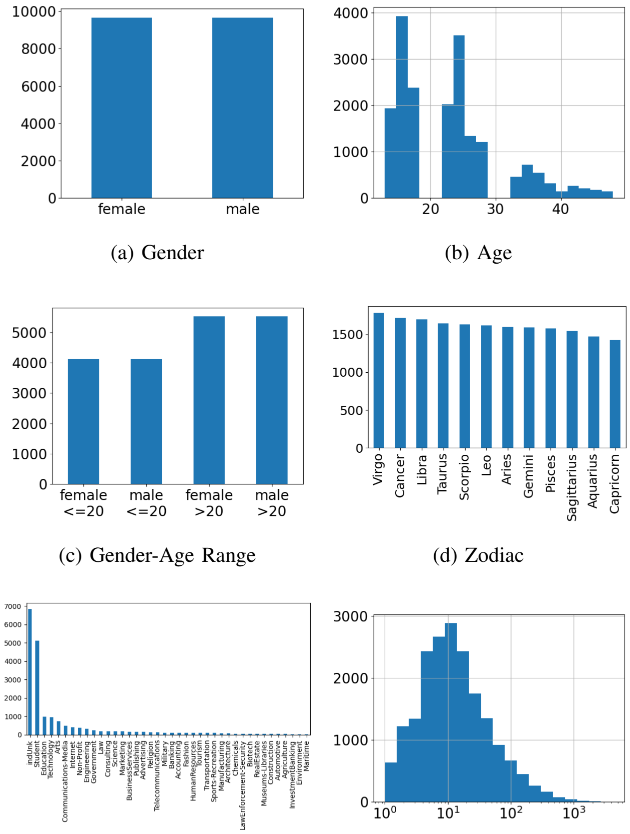
III. Research Design

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 by Fig. 1, which is created by Python packages `Pandas` and `Matplotlib`. 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) Zodiac: The distribution over zodiac is reasonable even.
4) Category: the most frequent category is unknown, which is trivial, while the second frequent one is student, far more than other categories.
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.



(a) Gender (b) Age

(c) Gender-Age Range (d) Zodiac

(e) Category (f) Number of Posts

Fig. 1. Data Overview. Histogram over (a) gender (b) age (c) gender and age range (d) zodiac (e) category (f) number of 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 Fig. 2, and the details of each step are described in the following subsections.

1) Data Cleaning: Before applying any text mining techniques, it is important to do basic data cleaning to improve data quality. In this step, a few operations for preprocessing will be carried out based on the observations of the dataset, with details as follows.

- **Problem**: At some place there is no whitespace between a punctuation and the word following it, which causes wrong tokenisation. Specifically, the punctuation might be tokenised with the following word as one token.
  Solution: Add whitespace after a punctuation if a word immediately follows it.
- **Problem**: Two or more consecutive quote symbols may cause wrong tokenisation.
  Solution: Replace two or more quotes as a double quote.
- **Problem**: The unicode quote may affect tokenisation and stopwords matching.
  Solution: Replace unicode quote by ASCII quote.
- **Problem**: The unicode quote may affect tokenisation and stopwords matching.
  Solution: Replace unicode quote by ASCII quote.
- **Problem**: Characters that are usually not part of normal English text may disturb tokenisation and POS tagging.
  Solution: Remove invalid characters such as "*“,”#", and so on.
- **Problem**: Sometimes people repeat a certain letter in a word for emphasis, but it will result in wrong words and also increase the vocabulary size.
  Solution: No English word has more than two consecutive appearances of the same letter, so three or more repetition of a letter is squeezed into two.
- **Problem**:
  Solution:

These operations are implemented by regex matching and substitution, or simple text replacing. To use regex, the Python's `re` package are imported.

2) Tokenisation: Tokenisation is usually the first step of all text mining pipelines, which includes sentence and word tokenisation. Sentence tokenisation is to split the whole text into sentences, while word tokenisation splits a sentence into word or tokens. In this project we use `nltk` package to do such task. This package provides two functions `sent_tokenize()` and `word_tokenize()` for both tokenisation. A document is first tokenised into sentences, and then each sentence is

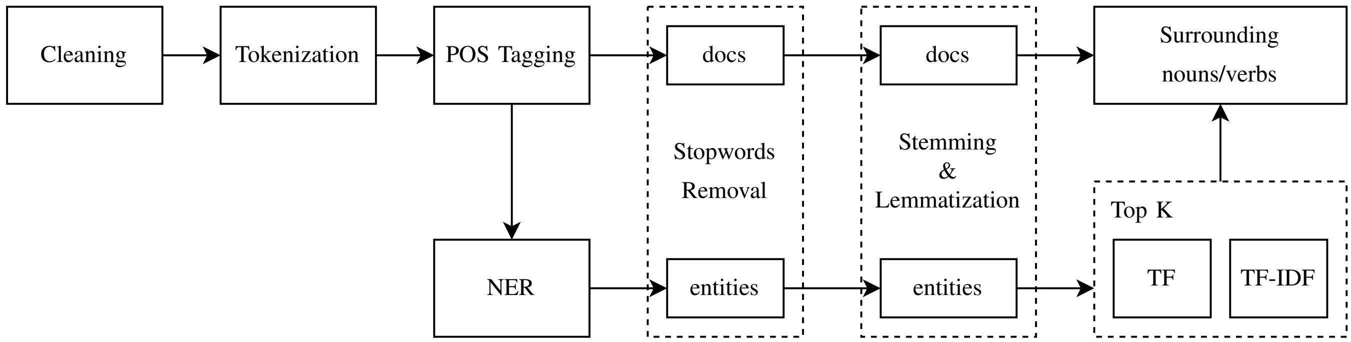


Fig. 2. Overall Architecture of Topic Mining Algorithm

tokenised into words. Finally, a document is represented as list of lists, as each sentence is a list of words.

3) POS Tagging: Part-Of-Speech (POS) tagging is the second step following tokenisation. In this step, each word is assigned by a POS tag. `nltk` provides a handy function `pos_tag()` to do this task. This function works on sentence level, and maps each word into a tuple which is the pair of word and POS tag.

4) Entity Extraction: In this project, a topic is defined as a “thing” or “object”. Therefore, in order to find the topics, we need to find all “things” or “objects” first. There are a few options to do this task, among which two methods will be employed by this project: Named Entity Recognition (NER) and parsing.

a) NER: Named entities are ideal candidates of topics as they denote real-world objects. `nltk` provides a function `ne_chunk()` to extract entities from sentences. The input of the function should be a list of tokens with POS tags, which is another reason why POS tagging should be done in previous step. The return value of this function is a list of chunks, each of which is basically a list and may contain a `label` attribute if it is a recognised entity. The entity type can be acquired by `label()` and the entity itself should be acquired by joining all the elements of the chunk.

b) Parsing: Another way to extract objects is parsing by pre-defined patterns. For example, it is reasonable to treat definite nouns as objects according to the grammar.

5) Stopwords Removal: Stopwords are most common words which carries no significant meanings. Removing stopwords can reduce the size of data to be proceeded as well as increase the result accuracy. `nltk` provides an out-of-the-box stopwords collection, but the experiment shows that some common words carrying no meaning are not included in the list. In order to expand the stopword list, more words are collected from website [1].

Stopwords removal is conducted after POS tagging and entity extraction because these two steps are sequence model, which means their performance rely on word order. If stopwords are removed before them, we will get sentences which do not comply with English grammar. In addition, stopwords removal are carried out on tagged documents as well as entities extracted. Theoretically, stopwords cannot be entity, but errors will happen in any POS tagging and NER model. Therefore, trying to remove stopwords can reduce the error introduced in previous steps.

6) Stemming and Lemmatisation: Stemming and lemmatisation are both techniques for text normalisation, that is, convert an inflected word into its root form. However, stemming and lemmatisation work in different way. Stemming removes suffix or prefix from a word, returning a word stem which is not necessarily a word. On the other hand, lemmatisation always looks for the lemma from word variations with morphological analysis. For example, stemming against the third-person singular form “flies” returns “fli”, while lemmatisation returns “fly”. In this project, these two methods are combined together to reach the maximum extent of word normalisation.

`nltk` provides various stemming algorithms such as `PorterStemmer` and `LancasterStemmer`, and one lemmatisation algorithm `WordNetLemmatizer`. In the code we use `WordNetLemmatizer` followed by `PorterStemmer`.

7) Word Count and TF-IDF: After all “objects” have been extracted and normalised, the next step is to find most popular ones as the most dominant topics. Popularity can be defined in various ways, and in this project two approaches are used: word count and Term Frequency-Inverse Document Frequency (TF-IDF). In the first method, we simply count the appearances of each entity and get the most two frequent ones. In the second method, we calculate the TF-IDF value of each entity word, following the definition

$$\begin{aligned}\text{TF-IDF}(t_i, d_j) &= \text{TF}(t_i, d_j) \times \text{IDF}(t_i) \\ &= \text{TF}(t_i, d_j) \log \frac{N}{\text{DF}(t_i)}\end{aligned}$$

$\text{TF}(t_i, d_j)$ is the Term Frequency of term t_i in document d_j, which is computed by count of t_i in d_j divided by the total number of terms in d_j. $\text{DF}(t_i)$ is the Document Frequency, which is the number of documents that contains t_i. As we can see here, TF-IDF is a term-document-wise number so a term has different TF-IDF values in different documents. In order to rank all terms over the whole dataset, TF-IDF values of

[1]https://gist.github.com/sebleier/554280

a term are averaged over all documents as the score of that term.

$$\text{score}(t_i) = \underset{d_j}{\text{avg}}\ \text{TF-IDF}(t_i, d_j)$$

Two different methods might return different results, which will be compared and analysed in section IV.

C. Evaluation Method

Evaluation of topics is challenging due to its nature of unsupervised learning. Among existing metrics, xx is chosen to evaluate the result of the methodology.

IV. Result, Analysis, and Evaluation

This section will show the results generated from the methodology described above, and provides evaluation and discussion of how good the topics are. Due to the large volume of the original dataset, the experiments were conducted with 5,000 and 10,000 documents randomly sampled out of the total 19,320 ones, and the results demonstrated consistency. Therefore, in the following part of this section, only results from 10,000 samples are presented.

A. Result

The results are displayed as word cloud generated by `wordcloud` package. Fig. 3 shows the topics mined by word count while Fig. 4 shows that by TF-IDF.

B. Analysis

C. Evaluation

V. Conclusion

This project has designed and implemented a complete solution to mine most popular topics from blogs. A variety of text mining technologies are employed and combined together to reach the goal. The results are compared and evaluated. Further and in-depth discussion is also provided.

VI. Open Issues and Future Works

There are still a few open issues remaining in the solution which can be improved by future work or changed if re-do this project.

fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn gdwgd fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn

fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn

fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn

fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn

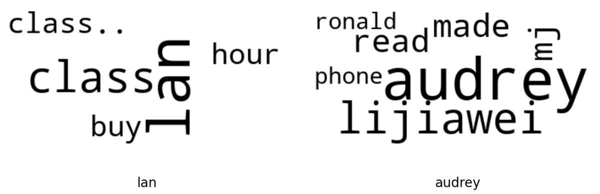
fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn

Fig. 3. Topics mined by word count. (a) male (b) female (c) 20 or younger (d) over 20 (e) everyone

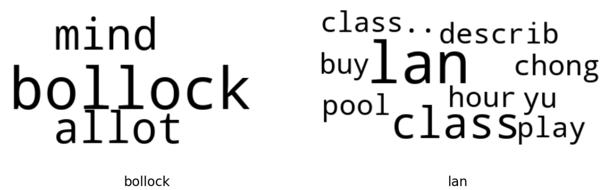
fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn

fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn fake wtdwt dwt wd

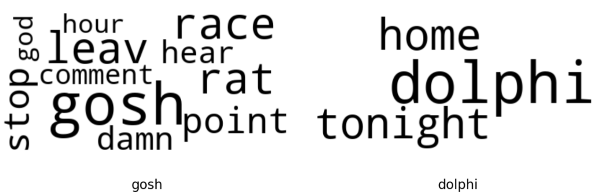
(a) Male



(b) Female



(c) 20 or younger



(d) Over 20

(e) Everyone

Fig. 4. Topics mined by TF-IDF. (a) male (b) female (c) 20 or younger (d) over 20 (e) everyone

twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn

fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn

fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn

fake wtdwt dwt wd twdt w d dw w wdt wd tw dtdw wd dwt d tagfdygafdgkdwlrn

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.

[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, J. Tan, 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,

Appendix
Source Code in Python

A. Code for topic mining

```
#!/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
NUM_SAMPLES = 10000
NUM_SAMPLES = 5000


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)

```

```python
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'])
```

```python
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)
```

```python
        dataset.append(rec)
    return dataset

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

leading_quote_re = re.compile(r'[\'\.~=*&^%#!|\-]+([a-zA-Z].*)')
def clean_word(word):
    if word in ("'ve", "'re", "'s", "'t", "'ll", "'m", "'d", "'", "''"):
        return word
    word = leading_quote_re.sub(r'\1', word)
    return word.strip()

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 = [clean_word(w) for w in nltk.word_tokenize(sent_str)]
                    sent = [w for w in sent if w != '']
                    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(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] /= df[token]

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

    print('-------------- result from TF ------------------')
    topics = get_top_topics(named_entities, n=50, method='tf')
    keywords = get_surroundings(topics, tagged_docs, n=num_keywords)
    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

def main_mine_topics(dataset=None, intermediate_data=None):
    if dataset is None:
        dataset = load_pkl('blogs.pkl')
    if intermediate_data is None:
        intermediate_data = load_pkl('intermediate_data.pkl')

    topics = {}
    topics['male'] = mine_topics(dataset, intermediate_data, group='male')
    topics['female'] = mine_topics(dataset, intermediate_data, group='female')
    topics['less_or_20'] = mine_topics(dataset, intermediate_data, group='<=20')
    topics['over_20'] = mine_topics(dataset, intermediate_data, group='>20')
    topics['all'] = mine_topics(dataset, intermediate_data, group='all')
    if _DEBUG:
        suffix = 'debug'
    else:
        suffix = date.today().strftime('%Y%m%d')
        if NUM_SAMPLES > 0:
            suffix += '-' + str(NUM_SAMPLES)

    save_json(topics, 'topics-{}.json'.format(suffix))

def main():
    if len(sys.argv) <= 1:
        phases = [1, 2]
    else:
        phases = [int(i) for i in sys.argv[1].split(',')]

    dataset = intermediate_data = None
    for ph in phases:
        if ph == 1:
            dataset, intermediate_data = main_intermediate()
        elif ph == 2:
            main_mine_topics(dataset, intermediate_data)

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

B. Code for analysis, evaluation and visualisation

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

import sys
import json
import numpy as np
import pandas as pd
from as2 import load_pkl, Record, MetaData, Post
import matplotlib.pyplot as plt
from wordcloud import WordCloud

MAX_FONT_SIZE = 80

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]
    df['char_count'] = [sum(len(p.text) for p in 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()))))

    plt.rcParams.update({'font.size': 20})
    df['gender'].value_counts().plot(kind='bar')
    plt.xticks(rotation=0)
    plt.gcf().tight_layout()
    plt.savefig('img/show-gender.png')

    plt.rcParams.update({'font.size': 10})
    plt.clf()
    df['category'].value_counts().plot(kind='bar')
    plt.gcf().tight_layout()
    plt.savefig('img/show-category.png')

    plt.rcParams.update({'font.size': 18})
    plt.clf()
    df['zodiac'].value_counts().plot(kind='bar')
    plt.xticks(rotation=90)
    plt.gcf().tight_layout()
    plt.savefig('img/show-zodiac.png')

    plt.rcParams.update({'font.size': 20})
    plt.clf()
    age = df['age']
    df['age'].hist(bins=20)
    plt.gcf().tight_layout()
    plt.savefig('img/show-age.png')

    plt.clf()
    cnt = df['blog_count']
    logbins = np.logspace(np.log10(cnt.min()),np.log10(cnt.max()), 20)
    cnt.hist(bins=logbins)
    plt.xscale('log')
    plt.gcf().tight_layout()
    plt.savefig('img/show-blog-count.png')

    plt.clf()
    cnt = df['char_count']
    logbins = np.logspace(np.log10(cnt.min()),np.log10(cnt.max()), 20)
    cnt.hist(bins=logbins)
    plt.xscale('log')
    plt.gcf().tight_layout()
    plt.savefig('img/show-char-count.png')

    plt.clf()
    df['gender_age'] = [g + '\n' + ('<=20' if a <= 20 else '>20') \
            for (g, a) in zip(df['gender'], df['age'])]
    df['gender_age'].value_counts()[[2, 3, 1, 0]].plot(kind='bar')
    plt.xticks(rotation=0)
    plt.gcf().tight_layout()
    plt.savefig('img/show-gender-age.png')


def color_black(word, *args, **kwargs):
    return '#000000'

def grey_color_func(word, font_size, position, orientation, random_state=None, **kwargs):
```

```python
    return 'hsl(0, 0%, {:d}%)'.format((MAX_FONT_SIZE - font_size) // (MAX_FONT_SIZE * 1))

def eval_topics(fpath, method='tf', top_k=2, num_words_in_topic=20):
    with open(fpath, encoding='utf8') as f:
        result = json.load(f)

    for group, topics2 in result.items():
        topics = topics2[method]
        for i, topic in enumerate(topics[:top_k]):
            topic_name = topic['topic']
            words = {}
            words.update(tuple(kw) for kw in topic['keywords'][:num_words_in_topic+1])
            if method == 'tf':
                words[topic_name] = topic['score']
            else:
                words[topic_name] = topic['keywords'][0][1] * 2  # fake frequency for display

            print('topic: ', topic_name, 'number of keywords:', len(topic['keywords']))
            wc = WordCloud(background_color="white",
                    max_font_size=80,
                    max_words=num_words_in_topic+1,
                    color_func=grey_color_func)
            wc.generate_from_frequencies(words)

            plt.clf()
            plt.imshow(wc, interpolation="bilinear")
            plt.axis("off")
            plt.title(topic_name, y=-0.25, fontsize=20)
            plt.gcf().tight_layout()
            fig_path = 'img/{}-{}-{}.png'.format(group, method, i+1, topic['topic'])
            print('drawing ' + fig_path)
            plt.savefig(fig_path)

def main():
    cmd = sys.argv[1]
    if cmd == 'show':
        show_summary(load_pkl('blogs.pkl'))
    elif cmd == 'eval':
        fpath = sys.argv[2]
        eval_topics(fpath, top_k=2)
        eval_topics(fpath, top_k=2, method='tfidf')

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