

CSE 788 - Data Mining and Knowledge Discovery

Mining Emotion-Word Correlations in a Large Blog Corpus

Project Report

Annatala Wolf and Manirupa Das



Spring 2009

Contents

1. Motivation.....	3
2. Problem Definition.....	3
3. Dataset.....	3
4. Theoretical Framework.....	4
5. Related Work.....	4
6. Proposed Work.....	4
Division of Responsibilities.....	4
Timeline.....	5
Expected outcomes.....	5
Challenges.....	5
Attempted Tasks.....	6
7. Methodology.....	7
Emotion Word Selection.....	7
Preprocessing.....	7
Storage.....	8
Indexing.....	8
Tractability.....	8
Implementation.....	8
8. Current Status.....	9
9. Sample Results.....	9
10. In Progress.....	10
11. Conclusions and Lessons Learned.....	10
12. Future Work.....	10
Appendix I.....	11
Appendix II.....	12

1. Motivation

Blogs constitute a useful, freely-available source of opinion and news data. The amount of blog data available on the Internet has grown at an exponential rate over the past few years. Many people are interested in the information contained in blogs, including users in the fields of market analysis, opinion polling, politics, sociology, and psychology. End-users are also interested in blog data as a source for networking, opinion data, and product reviews.

There are a number of unique features common to blog data. Blogs are usually user-generated, featuring content written with informal speech, abbreviations, neologisms, and syntax errors. Furthermore, many blog entries have a distinct emotional quality. These aspects of blogs can make information retrieval and analysis of blog features a challenging task, but they can also provide a unique insight into the users and communities which generate blog data. Mood information in particular is interesting because a user's mood may be reflected in several ways in informal speech: in the words a user chooses, in the length of a post, or in the topic area discussed, to name a few.

2. Problem Definition

Blog data presents us with unique challenges. Informal speech is less structured, and is harder to make sense of with traditional methods. Users express ideas differently in blogs e.g. emoticons, neologisms, and memes. Additionally, the size of available data sets can be very large.

We would like to know more about the words people choose when expressing themselves in a blog environment. Specifically, we are interested in identifying which, if any, correlations exist between words from specific semantically-related categories drawn from a basic theory on emotion. As an exploratory study, we will determine whether the words people choose correlate well with these categories. If successful, we could use this information to better predict how blog entries might cluster based on emotion, leading to improved models of information retrieval for blogs, or a better understanding of theoretical models of emotion in the unstructured literary domain.

3. Dataset

We used the dataset made available by Spinn3r.com for the ICWSM 2009 data challenge. This is a set of 44 million blog posts that covers a 62-day span between August 1st and October 1st, 2008. It is formatted in XML and further arranged into tiers approximating search engine ranking. As the tier information was not well-defined, we did not consider this as a feature in our study. The total size of the dataset is 142 GB uncompressed, or about 30 GB compressed. It spans some big news events such as the Olympics, both US presidential nominating conventions, the beginnings of the financial crisis. In order to eliminate spam and non-blog data, we only consider articles from six domains in the corpus known to primarily consist of personal blog entries: MySpace, WordPress, LiveJournal, BlogSpot, Vox and TypePad.

4. Theoretical Framework

We used the Ekman (1972) model of six basic emotions: anger, disgust, fear, happiness, sadness, and surprise. WordNet was used to expand the names of each of these categories into groups of related words. We carefully limited the size of the groups to form disjoint emotion-word clusters for each category. We then processed and analyzed the data to examine relationships between these words, as described in Problem Definition.

5. Related Work

Ha-Thuc et. al. (2009), Meeyoung et. al. (2009), Sood et. al. (2009), and Gordon et. al. (2009) describe the features of the ICWSM 2009 blog challenge corpus with various experiments conducted using the blog dataset, which was useful to us in identifying the steps necessary to preprocess and manipulate the dataset. Sood et. al. (2009) describes the process of clustering of emotion words into three mood categories. In future work this may be useful as an alternate method for grouping emotion words; we may later perform a comparison between our approach and the clustered model produced by Sood et. al. Depending on how far we get in our research with this dataset, we will consider approaches such as Latent semantic analysis, Deerwester et. al. (1990), that may be useful in indexing of the corpus by generating mood concepts from emotion and non-emotion words.

6. Proposed Work

Division of Responsibilities

Both Manirupa and Anna contributed in general to the overall design, structure, presentation, and report as required for the class exercise.

Annatala Wolf was responsible for identifying the theoretical approach for the study, identifying the initial feature set from the developed sets of mood words and document frequencies, designing the structure of the preprocessed file and index, and writing code to preprocess the files, index the preprocessed files into a compressed set of term bit-vectors (one per document), and the correlational mining.

Manirupa Das was responsible for developing the sets of mood words in a principled fashion, migrating the dataset to the OSC network, developing the bash scripts for running the entire dataset through preprocessing, indexing, and data mining, negotiating our special data storage needs with the OSC group, investigating the utility of running the system under a map-reduce framework, writing code to identify emotion words in preprocessed files and to generate document frequencies for the corpus, and writing most of the supporting documentation.

Timeline

The timeline for the indexing step was 3 weeks from project proposal. Data mining was expected to take 2 weeks. Given the exploratory nature of this study, a lot of the data mining tasks and tests were expected to depend on successful completion of the index. We planned to run tests on mood, examine associations, and analyze the results, which could potentially involve rerunning tests. If time permitted, we also planned to construct a classifier to test whether the associations we had mined were useful in predicting emotion category with the original emotion word features removed.

Expected outcomes

Expected outcomes included:

- The identification of a few consistent novel associations between the sets of mood words and other blog features. For example, it may be the case that “sad” posts tend to be shorter in overall length than posts of other types. (Although we did not analyze post length in this study, we considered this initially and will examine this as a feature in future work.) These kinds of associations may be useful in identifying the emotional quality of a text with limited features available, which is often the case in short blog texts.
- We predict that the emotion words within the same category will co-occur more frequently with one another, and less frequently with emotion words from other categories. We further predict that words from the comparison set will not show as strong of an association as the emotion words. If this prediction is false, it may be the case that two or more categories of emotion words are not as likely to be “lexically distinct” in blog texts.
- If time permitted, we intended to build a classifier to predict the presence of mood words on the basis of other features. We expect that this classifier would perform above chance levels.

Challenges

The articles in the dataset were well structured but minimally labeled (no POS tagging, topic identification, etc.). Some articles contained spam (most spam had been eliminated); others were cut off prematurely. Articles were sourced from many WWW domains, and not all articles were personal blogs: some were news sources, advertisement streams, etc. We knew in advance that computation was going to be either CPU or I/O intensive and efficient strategies needed to be employed to make using the dataset a tractable problem within the timeframe allowed.

Our initial plan was to set up Apache Hadoop on multiple nodes on a cluster to exploit the massive data parallelism for this blog corpus, and to run preprocessing, indexing and rule mining on this setup. We performed a preliminary install of the Hadoop software on a single node having 4 processors, on the cluster at OSC, and got some initial examples running for word counts on documents. The Hadoop

Distributed file system (HDFS) has a software installation requirement for each of the nodes that it is supposed to run on, to execute a map/reduce operation. However, due to the batch scheduling system for the OSC cluster, where the nodes assigned to a particular job are not known a priori, it was not possible to get required software installed on only a subset of the nodes as required by HDFS, as the Hadoop job may not finally run on these nodes. Also, it was not practical and administratively feasible to install the software on each node, as this is a large cluster that relies on a sophisticated scheduler to fairly balance loads. The workaround to this would be to use a variation called Hadoop-On-Demand that installs required software on the fly, on each of the nodes assigned to a job, and then starts executing the job. Again, due to administrative and time constraints, this was not a feasible option to evaluate at this time.

The other option for parallelizing some of our operations was to use PBS scripts. However in this case, separate shell scripts would have to be written to merge back partial results from the partial datasets processed by a cluster node. The performance gain would have been offset by the cost of merging and verification. Hence, the easiest option at this point that guaranteed results, given our project schedule, was to have shell scripts that were ready to execute most of the preprocessing code, and to have dedicated nodes to run these on, with sufficient disk and file count quotas, and to execute one run per tiered data subset at a time.

Initially we stored each article in a separate file, but upon hitting one million files we reached the user limit for file storage. This method had been selected because we were planning to use LingPipe for analysis, and storing each article in a separate file is a convenient way to feed articles to the system. However, the number of I/O requests became overwhelming. We realized from this experience that more thought needed to go into the process of storing the data between preprocessing and indexing. Our solution was to simply concatenate the preprocessed files with newlines between each article, and this sped up performance considerably. Thus we carried out all of our runs for preprocessing, indexing and some of our initial mining, on a tier-by-tier basis and then combined the outputs.

Attempted Tasks

We started out with the following basic tasks, with no emphasis at this stage, on parallelizing any of the implementation:

- Feature selection by emotion category word selection from WordNet
- Data preprocessing to extract articles containing emotion words from our categories
- Index creation and compression on the pre-selected articles
- Word Association rule mining on the compressed index

7. Methodology

The initial preprocessing was to be done using a feature list of emotion words obtained from the WordNet system, for each of our six emotion categories. In this step we extracted the articles of interest (those with at least one emotion word, in the correct web domains, marked as English, and at least 250 characters long after HTML stripping). We then changed words to lowercase (this was done incorrectly and had to be done again during indexing), removed stop-words, and used Porter stemming. Duplicates were then removed (also repeated at indexing).

Using the preprocessed dataset and a dictionary of the top-occurring words in the corpus, our plan was to construct a vector-based index which could be used to perform association rule mining. Each vector contains information on a single preprocessed article. There were several issues related to scale, that were an important factor that affected our design. E.g. it would be nice if relevant index data fit into core memory, though this was not necessary – however, it must certainly fit in disk quota. Also, the index must be constructible and accessible in a reasonable amount of time.

Emotion Word Selection

WordNet is a lexical database for the English language. It groups English words into sets of synonyms called synsets, provides short, general definitions, and records the various semantic relations between these synonym sets. We used the WordNet Search facility to select our initial set of features for each of our six emotion categories. So, for instance, given an emotion category such as Surprise, we expand it into each sense of the word, e.g. noun and verb senses, in this case. For each form of the noun sense of the word, we expand to the full hyponyms and consider only those that have a sense number of 2 or less, to minimize the distance from the root emotion word. We do the same for each verb sense of the word, which we expanded to the full troponyms. We obtained about 40 words on an average for each emotion category. A snapshot of this is presented in Appendix I

Preprocessing

For article selection, all blog entries (articles) were examined in the preprocessing step. We removed all the non-English articles. We retained only articles from six website domains which are known to be blog-related, viz. MySpace, WordPress, LiveJournal, BlogSpot, Vox and TypePad. After stripping HTML codes, we kept only those articles that were greater than 250 characters in length (a technique we borrowed from related work). Thus, no prematurely clipped articles or overly-short articles were retained.

For preprocessing of the features, all stop-words were removed (the, one, am, I, etc.), using the SMART-2 stop-word list. All duplicate words were removed. We only considered the presence or absence of a word in the presence of other words. Additionally, articles which did not contain emotion words were not retained since these were the only associations of interest.

Storage

The total preprocessed article count was 2,505,762! We had originally planned to use LingPipe for processing. For convenience of import to LingPipe, each article was stored as a separate file. But this led to millions of files in our initial tests with preprocessing, which meant wasted space from file slack, plus enormous I/O overhead to load files. Realizing we didn't need this structure, individual article files were combined into much larger article collections, where thousands of articles were contained in a single file.

Indexing

We set a minimum cutoff for words based on word count over a small subset of the corpus. Words are only included in the "dictionary" the index maps if: they appear in 0.01% of the index, or they are emotion words. Articles with no indexable words are not included in the index (thus only about 10% of articles are kept). The Indexer constructs a bit vector for each article, where each bit stands for the presence (1) or absence (0) of a word in that document.

To keep the index a reasonable size, we implemented our own version of run-length compression. Each byte starts with a zero-flag: If the flag is "0", the next seven bits are the next seven dictionary-indexed words in the article. If the flag is "1", the next seven bits record how many bytes worth of "00000000" this actually stands for. This can be easily read and manipulated without decompressing, which provides a great benefit to performance.

Tractability

Preprocessing took several days. The indexing process took only a few minutes. The final size of the index is about 418MB. After trimming, the index contains around 200,000 articles, which met with our expectations. Data mining takes a few hours, but is difficult because parameters must be tuned. If min support and confidence are too high then there are no results after the long wait. Debugging the mining code was also time-consuming.

Implementation

We wrote various programs and scripts using the Java and Python programming languages, and Unix shell scripting, to carry out the preprocessing, indexing, and analysis for our dataset. In particular, we wrote programs for preprocessing of the corpus for article selection using emotion words, document word frequency generation, compressed bit vector indexing of the preprocessed data, association rule mining (a simplified version for two-word correlations), and shell scripts for executing these on the OSC cluster. Some sample code snippets and scripts are provided in Appendix II.

8. Current Status

Steps completed so far include:

- Preprocessing of the entire corpus
- Emotion word selection through WordNet
- Feature selection by document frequency
- Index generation, with compression
- Association rule mining program completed (runs without decompressing index)

We are yet to implement the following, which we could not attempt due to time constraints:

- Parallelize the scripts for test runs for the preprocessing and association rule mining
- Build a classifier to verify our hypothesis and our discovered associations
- Compare results based on our theoretical model vs. the clustered model by Sood et. al.

We also need to verify the work we have done so far and extend the association rule mining to a more generalized approach.

9. Sample Results

We ran the association rule mining with a minimum support of >1% and a confidence of >10%. We ran this for 1,350 articles and obtained 114 associations, as illustrated in the snapshot below:

```
happy.out - WordPad
File Edit View Insert Format Help
complac:disconsol = (20% Confidence, 1% Support)
beatitud:gaieti = (20% Confidence, 1% Support)
beatitud:contrit = (20% Confidence, 1% Support)
felic:rueful = (27% Confidence, 1% Support)
jocular:disconsol = (20% Confidence, 1% Support)
jocular:bombshel = (21% Confidence, 1% Support)
dolor:disconsol = (36% Confidence, 1% Support)
penit:poignanc = (27% Confidence, 1% Support)
disconsol:plaintiv = (33% Confidence, 2% Support)
disconsol:blip = (22% Confidence, 1% Support)
disconsol:galvanis = (26% Confidence, 1% Support)
plaintiv:galvanis = (22% Confidence, 1% Support)
lugubri:bombshel = (26% Confidence, 1% Support)
bombshel:thunderclap = (22% Confidence, 1% Support)
For Help, press F1 NUM
```

10. In Progress

We are currently trying to mine the entire index. The initial few attempts have produced no association rules, probably because support and/or confidence are set too high.

Analysis of the results: Following are some of the questions to be asked...

- Do emotion words from all categories co-occur in expected clusters?
 - This may provide evidence to support or reject models of mood in informal speech
- What non-emotion words co-occur in the presence of emotion words?
 - We may be able to use this data to better cluster emotion words for information retrieval

11. Conclusions and Lessons Learned

The following are some of our conclusions and lessons learned from our experiences with this dataset:

- Large, unstructured blog data is particularly challenging to work with
- For tractability, how you plan to store the data is equally important as the computational approach
- Features can be difficult to select and must be selected carefully – reprocessing is very expensive
- Analysis requires forethought and testable predictions
- The compression method was easy to implement from scratch, and worked very well!

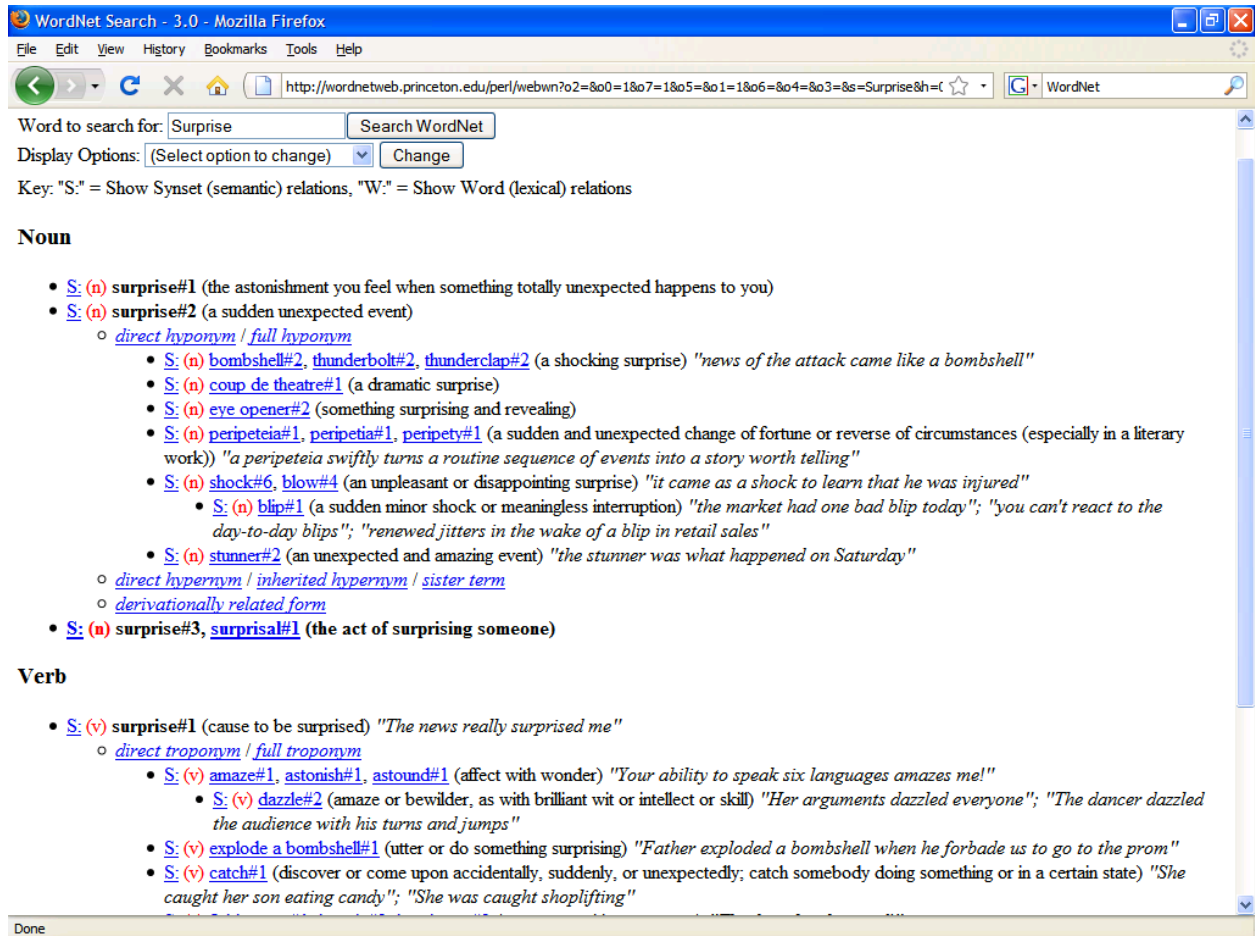
12. Future Work

Once our method is validated, we will finish mining on the entire index, with several support/confidence settings. We will perform analysis of our findings by comparing co-occurrence statistics across the six emotion categories. We will use the comparison group of non-emotion categories, to see if clustering is similar for non-emotion words. We may reprocess the corpus with less strict requirements now that we know process is tractable and scalable. We will also mine & analyze longer (multi-feature) associations. To do this, we will look at ways to efficiently employ appropriate ontologies for mood and emotion categories that may lead to considerable savings in space and computation, as we will use the pair-wise associations obtained in our initial runs to mine for additional associated features using the ontology, instead of the index. We will look at using WordNet for this purpose and also employ some of the existing command line tools and APIs provided within that system.

Alternatively, we may use ontologies for feature selection and verification of any classifiers we may construct. Additionally, now that we have working code for indexing and rule mining, we hope to configure an Apache Hadoop setup on a cluster and adapt our programs appropriately so as to be able to perform multi-feature rule mining in a map/reduce fashion to considerably speed up computation.

Appendix I

Snapshot of Emotion word category expansion using WordNet:



WordNet Search - 3.0 - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://wordnetweb.princeton.edu/perl/webwn?o2=&o0=1&o7=1&o5=&o1=1&o6=&o4=&o3=&s=Surprise&h=c

Word to search for: Search WordNet

Display Options: (Select option to change) Change

Key: "S." = Show Synset (semantic) relations, "W." = Show Word (lexical) relations

Noun

- [S: \(n\) surprise#1](#) (the astonishment you feel when something totally unexpected happens to you)
- [S: \(n\) surprise#2](#) (a sudden unexpected event)
 - [direct hyponym / full hyponym](#)
 - [S: \(n\) bombshell#2, thunderbolt#2, thunderclap#2](#) (a shocking surprise) "*news of the attack came like a bombshell*"
 - [S: \(n\) coup de theatre#1](#) (a dramatic surprise)
 - [S: \(n\) eye opener#2](#) (something surprising and revealing)
 - [S: \(n\) peripeteia#1, peripetia#1, peripety#1](#) (a sudden and unexpected change of fortune or reverse of circumstances (especially in a literary work)) "*a peripeteia swiftly turns a routine sequence of events into a story worth telling*"
 - [S: \(n\) shock#6, blow#4](#) (an unpleasant or disappointing surprise) "*it came as a shock to learn that he was injured*"
 - [S: \(n\) blip#1](#) (a sudden minor shock or meaningless interruption) "*the market had one bad blip today*"; "*you can't react to the day-to-day blips*"; "*renewed jitters in the wake of a blip in retail sales*"
 - [S: \(n\) stunner#2](#) (an unexpected and amazing event) "*the stunner was what happened on Saturday*"
 - [direct hypernym / inherited hypernym / sister term](#)
 - [derivationally related form](#)
- [S: \(n\) surprise#3, surprisal#1](#) (the act of surprising someone)

Verb

- [S: \(v\) surprise#1](#) (cause to be surprised) "*The news really surprised me*"
 - [direct troponym / full troponym](#)
 - [S: \(v\) amaze#1, astonish#1, astound#1](#) (affect with wonder) "*Your ability to speak six languages amazes me!*"
 - [S: \(v\) dazzle#2](#) (amaze or bewilder, as with brilliant wit or intellect or skill) "*Her arguments dazzled everyone*"; "*The dancer dazzled the audience with his turns and jumps*"
 - [S: \(v\) explode a bombshell#1](#) (utter or do something surprising) "*Father exploded a bombshell when he forbade us to go to the prom*"
 - [S: \(v\) catch#1](#) (discover or come upon accidentally, suddenly, or unexpectedly; catch somebody doing something or in a certain state) "*She caught her son eating candy*"; "*She was caught shoplifting*"

Done

Appendix II

Following are snippets from some of our code listings:

Dataset Preprocessor:

```
public static void main(String[] args) {
    try {
        int total = 0;
        int good = 0;
        File inDir = new File(args[0]);
        File outDir = new File(args[1]);
        outDir.mkdirs();
        FileWriter outStream = new FileWriter(outDir.getAbsolutePath()
            + "/" + "total.out");
        String[] listing = inDir.list();
        for (int i = 0; i < listing.length; i++) {
            FileInputStream inStream = new FileInputStream(
                inDir.getAbsolutePath() + "/" + listing[i]);
            inStream.read();
            inStream.read();
            BufferedReader input = new BufferedReader(
                new InputStreamReader(
                    new CBZip2InputStream(inStream)));
            String line = null;
            String title = null;
            boolean skip = false;
            line = input.readLine();
            while (line != null) {
                if (line.startsWith("<item>")) {
                    total++;
                    skip = false;
                }
                if (line.startsWith("<title>") && !skip) {
                    while (line.endsWith("</title>") == false) {
                        line = line + " " + input.readLine();
                    }
                    title = line.substring(7, line.length()-8);
                    title = replaceHTML(title);
                }
                if (line.startsWith("<dc:source>") && !skip) {
                    if (!line.matches(".*[myspace\\.com|wordpress\\.com|
                        livejournal\\.com|blogspot\\.com|vox\\.com|
                        typepad\\.com].*")) skip = true;
                }
                if (line.startsWith("<dc:lang>") && !skip) {
                    if (!line.startsWith("<dc:lang>en<")) skip = true;
                }
                if (line.startsWith("<description>") && !skip) {
                    line = line.substring(13);
                    while (line.endsWith("</description>") == false) {
                        line = line + " " + input.readLine();
                    }
                    line = line.substring(0, line.length()-14);
                    line = replaceHTML(line);
                    int metric = line.length();
                    line = parseWords(title + " " + line);
                    if (metric >= 250 && line != null) {
                        good++;
                        if (good % 1000 == 0) System.out.println(i + ":" +
                            good);
                        outStream.write(line + "\n");
                    }
                }
                line = input.readLine();
            }
            input.close();
        }
    }
}
```

```

        inStream.close();
    }
    outputStream.close();
    System.out.println(good + "/" + total);
} catch (IOException e) {
    System.out.println(e);
}
}
}

```

Shell script for preprocessing:

```

prefix=$HOME/class/788-DM/data/corpora/blogs/icwsm2009
output=$HOME/class/788-DM/data/corpora/blogs/output

for grp in 12 5 4 8 11 7 6 9 10 3 2 13 none
do
i=0
tg="tiergroup-"$grp
echo $tg
for folder in `ls $prefix/$tg`;
do
echo $folder
for day in `ls $prefix/$tg/$folder`;
do
echo $day
i=`expr $i + 1`
java -jar blogParser2.jar $prefix/$tg/$folder/$day $output/$tg/
mv $output/$tg/total.out $output/$tg/$i".out"
done
done
done

#--#

```

Preprocessed output directories statistics:

Consolidated output file sizes by tiergroup:

```

$ du -sm tiergroups/* | sort -nr
1834 tiergroups/tg-none.out
748 tiergroups/tg-1.out
140 tiergroups/tg-13.out
16 tiergroups/tg-2.out
10 tiergroups/tg-10.out
7 tiergroups/tg-9.out
6 tiergroups/tg-7.out
5 tiergroups/tg-6.out
5 tiergroups/tg-3.out
4 tiergroups/tg-8.out
3 tiergroups/tg-11.out
2 tiergroups/tg-5.out
2 tiergroups/tg-12.out
1 tiergroups/tg-4.out

```

Output file article counts by tiergroup:

```

$ wc -l tiergroups/* | sort -nr
2505762 total
1626663 tiergroups/tg-none.out
690781 tiergroups/tg-1.out
140911 tiergroups/tg-13.out
13426 tiergroups/tg-2.out
8155 tiergroups/tg-10.out
6098 tiergroups/tg-9.out
4131 tiergroups/tg-3.out
3586 tiergroups/tg-7.out
3549 tiergroups/tg-6.out

```

```
3093 tiergroups/tg-8.out
2586 tiergroups/tg-11.out
1355 tiergroups/tg-12.out
1106 tiergroups/tg-5.out
322 tiergroups/tg-4.out
```

Document word frequency generator:

```
#!/usr/bin/python
#
#usage: ./docfrequency.py <article_file> <frequencies_hash_file>
#
import sys

file = sys.argv[2]
#open frequencies file
f = open(file, 'rb')
c = f.read()
f.close()

#load into dict
c.strip()
d = eval(c)

#find article word frequencies
fdata = open(sys.argv[1], 'r')
txt = fdata.read()
list = txt.split()
for w in list:
    w = w.lower()
    if w in d.keys():
        d[w] += 1
    else:
        d[w] = 1

#write out frequencies file
f = open(file, 'wb')
f.write(str(d))
f.close()

#--#
```

Emotion word counter:

```
#!/usr/bin/python
#
#usage: ./get_emotion_wordcounts.py <emotion_word_file> <input_frequency_file>
<output_frequency_file>
#
import sys

#open emotion words file
f = open(sys.argv[1], 'rb')
e_list = f.readlines()
f.close()

#open frequencies file to read
f = open(sys.argv[2], 'rb')
f_list = f.read()
f.close()

f_list.strip()
d = eval(f_list) #load input frequencies list into dictionary

d_em = {} #initial dict for emotion words
for k in e_list:
    e = k.strip()
```

```

    if e in d:
        d_em[e] = d[e]

#write out emotions dict(frequencies) to file
f = open(sys.argv[3], 'wb')
f.write(str(d_em))
f.close()

#--#

```

Indexer:

```

public static void putVector(FileOutputStream out, boolean[] array)
    throws IOException {
    ArrayList<Byte> compressed = new ArrayList<Byte>();
    byte current = (byte) 0;
    int emptySeptets = 0;
    for (int i = 0; i < array.length; i++) {

        // Update the current bit
        boolean bit = array[i];
        if (bit) {
            current |= (byte) (0x1 << (6 - i%7));
        }
        // If this bit ends a big stream of zeroes, flush compress output
        if (current != 0 && emptySeptets != 0) {
            compressed.add((byte) ((byte) emptySeptets | (0x1 << 7)));
            emptySeptets = 0;
        }
        // If we've finished a septet and there's a bit in it, output
        if (current != 0 && (i+1)%7 == 0) {
            compressed.add(current);
            current = 0;
        }
        // But if the septet was empty, increment count
        else if (current == 0 && (i+1)%7 == 0) emptySeptets++;
        // If we hit max septets, flush compress output
        if (emptySeptets == 127) {
            compressed.add((byte) ((byte) emptySeptets | (0x1 << 7)));
            emptySeptets = 0;
        }
    }
    // Final flushing
    if (emptySeptets > 0) {
        if ((array.length % 7) > 0) emptySeptets++;
        compressed.add((byte) ((byte) emptySeptets | (0x1 << 7)));
    }
    if (current != 0) {
        compressed.add(current);
    }
    if (emptySeptets == 0 && current == 0 && (array.length % 7) > 0) {
        compressed.add((byte) ((byte) 0x81));
    }
    // Write to file
    for (byte b : compressed) {
        out.write(b);
    }
}

```

Association rule miner:

```
public static void main(String[] args) {
    try {
        NumberFormat perForm =
            NumberFormat.getPercentInstance();
        double support = Double.parseDouble(args[3]);
        double minConf = Double.parseDouble(args[4]);

        // Read in the word list
        BufferedReader keys = new BufferedReader(new FileReader(args[0]));
        HashMap<Integer, String> wordMap = new HashMap<Integer, String>();
        String keyWord = keys.readLine().toLowerCase();
        int wordCount = 0;
        while (keyWord != null) {
            keyWord = keyWord.toLowerCase();
            if (!wordMap.containsKey(keyWord)) {
                wordMap.put(wordCount, keyWord);
                wordCount++;
            }
            keyWord = keys.readLine();
        }
        keys.close();
        System.out.println(wordCount + " total features considered.");

        // Mine the index
        FileInputStream input = new FileInputStream(args[1]);
        int[][] matrix = new int[wordCount][wordCount];
        boolean[] line = new boolean[wordCount+8];
        int articles = 0;
        int counter = 0;
        while (input.available() != 0) {
            byte b = (byte) input.read();
            if (b < 0) {
                counter += (b & (byte) 0x01111111);
            }
            else {
                line[counter++] = ((b & (byte) 0x01000000) != 0);
                line[counter++] = ((b & (byte) 0x00100000) != 0);
                line[counter++] = ((b & (byte) 0x00010000) != 0);
                line[counter++] = ((b & (byte) 0x00001000) != 0);
                line[counter++] = ((b & (byte) 0x00000100) != 0);
                line[counter++] = ((b & (byte) 0x00000010) != 0);
                line[counter++] = ((b & (byte) 0x00000001) != 0);
            }
            if (counter >= wordCount || input.available() == 0) {
                for (int i = 0; i < wordCount; i++) {
                    for (int j = i; j < wordCount; j++) {
                        if (line[i] && line[j]) {
                            matrix[i][j]++;
                        }
                    }
                }
                counter = 0;
                articles++;
                line = new boolean[wordCount+8];
                if (articles % 100 == 0) System.out.println(articles);
            }
        }
        input.close();
        System.out.println(articles + " total articles read from index.");

        double size = (double) articles;
        int associations = 0;
        System.out.println("Index read complete.\n");
        System.out.println("Now outputting associations...");
        // Output the matrix information
        PrintWriter output = new PrintWriter(new FileWriter(args[2]));
        for (int i = 0; i < wordCount; i++) {
            for (int j = i+1; j < wordCount; j++) {
```