Eudora MoodWatch Architecture
March 7, 2006

1 Introduction

Eudora uses a text processing engine to analyze the content of mail messages in an effort to identify offensive, inflammatory or aggressive language. The engine does not distinguish between kinds of offensive language but simply returns an overall offensiveness score which is then assigned to the corresponding message. The score is sometimes humorously referred to as the flame level.

This tool is used in several ways in Eudora. It can be used to warn a user if a message they are composing or sending has language that might be construed by the recipient as offensive. It can also be used to indicate to the user that a message they have received has potentially offensive language before they view the contents of the message.

At the heart of the MoodWatch feature is a general text analysis engine that processes text based on a set of provided dictionaries and returns a score.

2 History of the Text Analysis Engine

The engine is based on a research project headed by David Kaufer, a professor in the English Department at Carnegie Mellon University. It was an attempt to provide a software tool to analyze text by matching words and phrases against a set of dictionaries. The original purpose of this project was to categorize a given text (e.g., rhetoric, news, fiction, etc.) based on its content and to automatically detect certain specific aspects of the text.

It should be noted that the engine itself is a generic engine for matching a text against a set of dictionaries. Any assumptions about the meaning, content or even the language of the text are rooted in the dictionaries provided. The engine itself could be used to analyze text on any scale provided you have the appropriate dictionaries.

The use of this engine for determining mood was made possible by extensive research on language used in various forms of electronic communication, notably newsgroups, especially those where offensive language is more common. This research was performed by David Kaufer. It was never intended that the dictionaries that we include with Eudora would remain constant but rather would be improved over time and adapted to account for changes in language usage. Although minor adjustments have been made to the dictionaries, the response that the MoodWatch feature received never seemed to warrant the time and effort involved in any major updating of the dictionary.

3 The Text Analysis Engine

The general purpose text analysis engine is at the heart of the MoodWatch feature.

3.1 Implementation

The original engine was coded in Java and was translated into C for purposes of use in Eudora. The code was translated as literally as possible and no effort was made to reorganize it. Had the code been written in C originally, doubtlessly the organization would be very different.

The text analysis engine (or TAE) is implemented in four files: TAE.c, TAECommon.c, TAEDictionary.c and TAEText.c. The engine is implemented in C is as generic a fashion as possible and the code is nearly identical between Macintosh Eudora and Windows Eudora. Where necessary, platform specific code is conditionally compiled.

The dictionary file is read into memory as a directed acyclical word graph and the phrase matching code traverses this graph to compare the specified text against the dictionary. The matching code finds the longest match possible in the dictionary. For example, if the phrases “got laid” and “got laid off” both appear in the dictionary then the phrase “I got laid off today” would match against the longer phrase “got laid off.”

3.2 Accessing the Engine

The following code excerpt is taken from the file TAE.h and illustrates the basic process for using TAE to analyze a text:

```
struct TAEDictState     taeds;
if (TAEInitDictionary(&taeds, "dict.dat"))
{
    struct TAESessionState     taess;
    if (TAEStartSession(&taess, &taeds))
    {
        // To keep the overall score:
        int                         iScore = 0;
        // Optional: Use this only if you want detailed score info.
        struct TAEScoreData          taesd;

        // Call this as many times as you like with different
        // sections of the text.  Don't pass in the same text more
        // than once or pass in overlapping portions of text or
        // the scoring will not be accurate.
        TAEProcessText(&taess, cText);

        // Get the score data.  Do this after you are done calling
        // TAEProcessText() and before you call TAEEndSession().
        // If you do not care about the detailed scoring data (most of
        // the time you won't) simply pass NULL for the second param.
        iScore = TAEGetScoreData(&taess, &taesd);

        printf("\n\nFlame level: ");
        switch (iScore)
        {
            case 3: printf("high\n"); break;
            case 2: printf("medium\n"); break;
            case 1: printf("low\n"); break;
            case 0: printf("none\n"); break;
        }

        TAEFreeScoreData(&taesd);
        TAEEndSession(&taess);
    }
    TAECloseDictionary(&taeds);
}
```

That describes the basic functioning of the TAE, including all functions that calling code would need to use.

3.3 Special Purpose Code

It was our intention to keep the engine itself general purpose to allow the same code to be used for a variety of purposes, not just MoodWatch. However, over time various scoring and exception code was added that made the engine more specific to MoodWatch. Those changes do not affect the overall engine function and could easily enough be removed or special cased to make the engine truly multi-purpose.

3.3.1 Special Scoring Code

The function TAE_AnalScore() takes the raw results of the dictionary matches and determines an overall score from 0 (inoffensive) to 3 (highly offensive). This code is obviously specific to MoodWatch and any use of the text engine would probably need some equivalent to this function to make sense of the raw dictionary match data.

3.3.2 Exceptions to the Dictionary

There is code in the engine which we added to allow for excluding of certain words that would otherwise match dictionary entries. Specifically, there are words which in certain contexts might be considered offensive, but are also proper names for people. An obvious example is “Dick.” It is obviously undesirable to mark these words as potentially offensive when they are presented as names. A quick work-around for this was to add code to the function TAEDictionary_CategorizeWordsAndPhrases() that will disregard a certain list of hard-coded words, in this case capitalized names. This isn’t an optimal solution and does go against the general purpose nature of the engine. It is also worth noting that since this exception handling code is not particularly optimized, it is desirable to keep the list of exceptions small.

4 The CMoodWatch Class

The TAE source code is encapsulated inside the C++ CMoodWatch class which makes it possible to use the MoodWatch feature without understanding the internals of the TAE. The CMoodWatch class has a simple interface and there are only a few methods of any interest to calling code. The following code shows how to use the CMoodWatch class to obtain a MoodWatch score on a text, including use of all interesting methods:

```
CMoodWatch moodwatch;
if (!moodwatch.Init())
{
    return;
}

TAEAllMatches taeamMatches;
while (/*there is more text to scan*/)
{
    strcpy(pText, /*text from desired source*/);
    moodwatch.AddText(pText, strlen(pText), &taeamMatches);
}

int iScore = moodwatch.GetScore();
```

The data returned in the taeamMatches variable contains data for every word or phrase in the text that matches one or more dictionaries, including the starting and ending point for where the phrase occurs in the text as well as the dictionary that the text matched. This data is used in composition windows to indicate any words or phrases the user typed which might be considered offensive.

5 The Dictionaries

The MoodWatch dictionary is actually composed of a number of sub-dictionaries. There are currently four sub-dictionaries incorporated into the main dictionary: high flame, low flame, safe and dummy. These are detailed in the descriptions of the files below.

5.1 Dictionary Source Files

tones.txt – List of all dictionary source files.

wordclasses.txt – List of wordclasses. Wordclasses can be thought of as wildcards or placeholders. For example, anywhere in the dictionary that the word “WAS” appears will cause dictionary entries to appear for the phrase using both “was” and “were” in its place.

H_DicWord*n*.txt/H_DicWord*n*_p.txt – The high flame dictionary files. This dictionary contains the more highly offensive phrases. The dictionary is broken up into three separate sets of files for easier managing.

NH_DicWord*n*.txt/NH_DicWord*n*_p.txt – The lower flame dictionary files. This dictionary contains the less offensive phrases. This dictionary is also broken up into three separate sets of files though only the first dictionary file currently contains anything.

SafeDic.txt/SafeDic_p.txt – The safe dictionary files. There are certain words and phrases that might seem offensive by themselves but are completely innocent when viewed in their full context. For example, the phrase “get laid” changes meaning completely when viewed in the context of “get laid off.” Since the matching code favors the longest match the potentially offensive phrase “get laid” will not match and the safe phrase “get laid off” will match. The scoring algorithm disregards any matches that occur in the safe dictionary.

DummyDic.txt/DummyDic_p.txt – The dummy dictionary files. A dummy dictionary included for debugging purposes only. You can fill the dummy dictionary with specific target phrases then test and make sure the engine correctly matches those phrases in the text to this dictionary. The dictionary that ships with Eudora obviously has an empty dummy dictionary.

5.2 Building the Dictionary

The dictionary builder is a Java program. It should be noted that building the dictionary is a resource intensive process and performing it on an underpowered machine will take an excessive amount of time. In the best case it will probably take more than an hour to build the current dictionaries, on an underpowered machine it can easily take days.

There are actually three different versions of the dictionary builder program and the differences between the three versions are detailed in a file named DictBuilderInfo.txt. Here is the explanation from that file:

> There are three built versions of the dict builder:
> Full-Heuristic-DictBuilder.jar - Full heuristics on with the extra check.
> Light-Heuristic-DictBuilder.jar - Only one heuristic in place which is the safest looking. No guarantees though.
> Brute-DictBuilder.jar - No heuristics. Slightly different because Brute was unnecessarily doing the early recursion, which for Brute was an orphaned node - no point in doing the recursion. (i.e. removing this might make Brute take slightly less time, but Heuristic more).
>
> In practice "full heuristic" was found to be incorrect/inconsistent (lost nodes?). Brute was no heuristics and should have been absolutely safe and correct, but it took too long to complete (Rob at one point said it was stuck - so perhaps it never completed). In the end we went with the "light heuristic" build.

Light heuristic corresponds to:
attemptDiamondExpansion = true;
attemptSmartCopy = false;

attemptDiamondExpansion - is the fairly innocent attempt to add all word class members without copying "My !pet is cute" where !pet is dog, cat, or snake creates a "diamond" where "My " and " is cute" is shared. There is code in place to check for collisions with preexisting words to avoid bad matches, etc.

attemptSmartCopy - is the more complicated attempt of when a collision has been detected to point back to another portion of the tree rather than copying huge amounts of the tree. Obviously something is awry with this code, because if the node count is off at the end (which is was for our bad builds) then not all nodes ended up being visited. This might be as simple as the flag I use to avoid revisitation is being set incorrectly in the smart copy cases.

5.3 Binary Dictionary File

The dictionary sources are compiled into a single binary file containing all of the phrases. This file is named flamelex.dat and is included by the installer. Details about the actual contents of this binary file are not important.

6 Scoring Messages

Messages are processed for mood and subsequently assigned a mood score at different times depending on their context.

6.1 Scoring Algorithm

The function TAE_AnalScore() contains the following table which is used for turning the number of matches in the text into a score:

```
short scores[9][7] = {/* H words   non-H        total       score */
                        {1, 100,    0, 0,       0, 0,             3},
                        {2, 1000,   0, 0,       0, 0,             3},
                        {1, 1000,   2, 100,     0, 0,             3},
                        {1, 500,    0, 0,       0, 0,             2},
                        {1, 1000,   1, 100,     0, 0,             2},
                        {0, 0,      3, 500,     0, 0,             2},
                        {0, 0,      0, 0,       2, 50,            2},
                        {0, 0,      0, 0,       2, 100,           1},
                        {1, 8000,   0, 0,       0, 0,             1}
                     };
```

The first pair of numbers in each row represents the minimum number of matches in the highly offensive dictionary and the minimum number of words in the text being analyzed. For example, 1, 100 means that there must be at least 1 match in the highly offensive dictionary for every 100 words. The second pair of numbers represents the minimum number of matches in the less offensive dictionary and the minimum number of words in the text. The third pair of numbers represents the minimum number of matches in either of the two dictionaries and the minimum number of words in the text. The final number indicates the score. It should be noted that this table is more a result of trial and error than any formal research data and the numbers could easily be adjusted to change the overall score.

6.2 Text to Score

It makes little sense to pass all of the message data (body and headers) through the mood scoring engine, only those parts that the user might reasonably composed by a user. This includes the body, the subject and the “From:”, “To:” and “Cc:” fields. The address fields are important because it is possible for the user to insert text into those fields that is not strictly part of an email address and could include offensive language.

6.3 On Composition

Composition messages are continually scanned for mood just as they are continually scanned for spelling errors. Details about how this score is conveyed to the user are discussed in the Interface section below.

6.4 On Arrival

When new messages arrive they are scanned for mood. As messages are being fetched from the spool summaries are created for those messages. During the process of creating the CSummary object associated with a message a mood score is generated for that message.

7 The Interface

The mood score of a message is presented to the user in a variety of ways depending on the context. The icons chosen to represent mood score are one, two or three chili peppers to indicate increasing levels of offensive language and (in some contexts) an ice cube to indicate a message with little or no offensive language.

7.1 The TOC View

The TOC view has a new column which shows the mood score of each message. This column displays one, two or three chili peppers to indicate the mood score. If a message is given a mood score of 0 no icon is shown rather than showing the ice cube.

7.2 Composing Mail

The mood score of a message being composed is conveyed in two ways: through underlining and through a composition window toolbar button.

7.2.1 Underlining

If the entire message is scored as potentially offensive (a score of 1, 2 or 3) then any words or phrases that match dictionaries are underlined. Text that matches the mild dictionary is underlined in a light green. Text that matches the hot dictionary is underlined in red and green (reminiscent of a chili pepper). If the entire message is given a score of 0 then no underlining is done, even if some text did match one of the dictionaries.

7.2.2 The MoodWatch Status Button

There is a button on the composition toolbar that indicates the overall mood score of the current message content. This button uses the ice cube and chili pepper icons to indicate the score.

7.3 Sending Mail

Eudora offers two ways to use the mood score when sending or queuing mail: a warning dialog and delayed sending/queuing.

7.3.1 Warning Dialog

When sending or queuing a message Eudora can optionally warn the user if the message is potentially offensive. There is an option that allows the user to specify what the minimum score needed before the user is warned, including an option to never warn the user.

7.3.2 Delayed Sending/Queuing

When sending or queuing a message Eudora can optionally delay the sending of a message for a period of time if the message is potentially offensive. As with the warning dialog, there is an option that allows the user to specify the minimum score needed before message sending is delayed. The user can also specify how long the delay should be, the default being 10 minutes.

8 Patent Information

Qualcomm has a patent application submitted for the MoodWatch technology. The application is:

#20020013692 Method of and system for screening electronic mail items (January 31, 2002)