

Text Analysis Engine Proposed 5.1 Changes

The Text Analysis Engine (the engine behind the MoodWatch feature) is a C translation of a Java prototype. The Java prototype was not designed to be a final product and numerous changes were required to make the translation suitable for inclusion in Eudora. Due to time constraints, as of Eudora 5.0 only about 2/3 of the redesign was truly complete. The following is my proposal for finishing the redesign as well as addressing issues that have arisen since incorporating the engine into Eudora. I am also proposing changes to the way we handle the dictionary building process.

For purposes of this discussion, there are two major components to the Text Analysis Engine: the tokenizer breaks the text down into meaningful tokens (words and punctuation) and the analyzer compares the tokens against the dictionary looking for matches.

I am proposing the following changes to the Text Analysis Engine. Details will be discussed below.

Minor changes:

Change the analyzer code to treat the internal data pointers as offsets (4 hours)

Add the ability to ignore “safe” text portions when scoring (4 hours)

Engine revamping changes:

Revamp the tokenizer (2 days)

Add HTML tag parsing (4 days)

Change the text analyzer to tokenize on the fly (3 days)

Dictionary builder changes:

Convert the dictionary builder from Java to C (8 days)

Make changes to the dictionary data file format (1 day)

Address dictionary builder heuristics problems (1 day)

Change the analyzer code to treat the internal data pointers as offsets

The binary MoodWatch dictionary consists of a collection of nodes and edges. The entire dictionary is read in as a single data block and internal references are used to allow the engine to navigate from an edge to the corresponding node within this data block. As of Eudora 5.0, these internal references are pointers to the actual memory locations of the nodes, but will be changed to be offsets from the beginning of the block. By changing these pointers to offsets we would allow the memory block to be moved. The only interface change would be to add two calls: `TAEGetDataPointer()` to get the current starting location of the memory block and `TAESetDataPointer()` to set a new starting location. All other changes would be internal to the engine and would be transparent to the calling code.

Add the ability to ignore “safe” text portions when scoring

A piece of text may be passed to the engine in several smaller pieces. The total number of tokens in each piece of text and the number of matching tokens for each piece of text are tallied and the score for the entire piece of text is derived from these data items. Adding the ability to subtract out the number of tokens in text with no matches will allow us to effectively ignore “safe” text. (Requested by Steve)

Revamp the tokenizer

The tokenizer is in need of an overhaul to address design issues and newly discovered problems:

- Simplify the tokenizer so that single quote handling is no longer a special case requiring allocation of a separate data item. The single quote handling is an artifact of the way the Java prototype was coded and this change has been planned from the beginning.
- Look into and fix any problems with the current tokenizer. I know for a fact that characters in the extended character set are treated as delimiters (for example, “appétit” is broken into “app” and “tit”). There have been results that suggest there may be problems with the treatment of some punctuation marks.

Add HTML tag stripping

The tokenizer should not count any part of an HTML tag as a valid token and thus no part of an HTML tag should have any bearing on the score.

Change the text analyzer to tokenize on the fly

As of Eudora 5.0, the text tokenizer is invoked twice, once to determine the number of tokens so we can allocate them in one block and once to fill the one block with the token data. This was done knowing that we would be sacrificing speed for the sake of reducing the number of memory allocations since the number of allocations was causing problems on the Mac. After tokenizing, the analyzer iterate through the list of tokens. Under the new plan, the analyzer would simply make one pass through the tokenizer asking for one token at a time, reducing the three loops down to a single loop.

Convert the dictionary builder from Java to C

The software that we use to convert the raw dictionary text files into the binary dictionary is currently implemented in Java. There are several motivations to make the move from Java to C:

- This would allow us to move ownership of all aspect of the binary dictionary building and reading to me. This way when I want a change to the dictionary file format I won't need to involve Geoff.
- This would provide a more consistent debugging environment for finding problems with the dictionary builder. Currently, the version of the dictionary builder that uses the full heuristics (the fastest way to build the dictionary) produces different output than the non-heuristics version.
- At some point we may want to incorporate the dictionary builder code into Eudora. One possible way to do this would be to have the user create a text file that Eudora would convert to a binary dictionary on the fly and have the analyzer use this dictionary in conjunction with the main dictionary.

Make changes to the dictionary data file format

With the dictionary builder converted to C, there are several file format changes we should make:

- Add a 256 byte buffer at the beginning of the file. The first two bytes would indicate the file format version number (as they do in the current format). The next two (four???) bytes would indicate the file revision number. This revision number would be independent of the file format version number and would

- indicate what revision of the text source files were used to build the dictionary. The remaining bytes would be a buffer for future data.
- For each sub-dictionary in the binary dictionary file add a 2 byte field to provide additional data about the given sub-dictionary. The raw text files used to create the binary dictionary are divided into several sub-dictionaries. At one time these different sub-dictionaries represented different categories of flames, but now the only distinction is between high and low flames. As of Eudora 5.0, each sub-dictionary has an entry in the dictionary file that does nothing more than indicate the presence of the sub-dictionary. We will add a field that will allow us to indicate additional data about each sub-directory. In the case of the flame dictionaries this could be used to specify whether a given sub-dictionary is a high flame or a low flame. As of Eudora 5.0, the data about whether a given sub-dictionary is a high or low flame must be inferred via the order in which the sub-dictionaries appear. Adding this additional data to the file format would allow us to remove any assumptions about sub-dictionary order from the scoring code.

Address dictionary builder heuristics problems

The version of the dictionary builder that uses the full heuristics (the fastest way to build the dictionary) produces different output than the non-heuristics version. We need to address these problems so that we can take advantage of the most optimized version of the dictionary builder.