Detecting flames and insults in text

Altaf Mahmud
Brac University, Dhaka, Bangladesh
altaf.mahmud@gmail.com

Kazi Zubair Ahmed
Brac University, Dhaka, Bangladesh
infernohere@yahoo.com

Mumit Khan
Brac University, Dhaka, Bangladesh
mumit@bracuniversity.ac.bd

ABSTRACT

While the Internet has become the leading source of information, it is also become the medium for flames, insults and other forms of abusive language, which add nothing to the quality of information available. A human reader can easily distinguish between what is information and what is a flame or any other form of abuse. It is however much more difficult for a language processor to do this automatically. This paper describes a new approach for an automated system to distinguish between information and personal attacks containing insulting or abusive expressions in a given document. In Linguistics, insulting or abusive messages are viewed as an extreme subset of the subjective language because of its extreme nature. We create a set of rules to extract the semantic information of a given sentence from the general semantic structure of that sentence to separate information from abusive language.

Key Words: insult, information, semantic, factive.

1. Introduction

Most of the time, Internet users get frustrated when they search for any information in a specific site, because some peoples take it as a fun to use personal attacking or insulting messages for on-line communication. One of the best examples can be 'wikipedia' (URL: http://www.wikipedia.org) where many times these occurrences are happened, which they called 'wiki vandalism'. Such vandalisms in wikipedia are subsequently reverted by another user. But, if an automated system would help a user for distinguishing flames and information in a web page or in e-mail, user can decide whether or not to read that article before. Some messages can contain insulting words or phrases but still they are considered as factual information. For example: a sentence '*X* is an idiot' is an insult, doesn't contain any factual information and should be discarded. But a sentence '*Y* said that *X* is an idiot' is not an insult any more, because it could conveys information about what *Y* said about *X*. Normal text searching methods or looking for obscene expressions will annotate both of them as flame. From this perspective, we outline a sophisticated sentence classification system using Natural Language Processing, to identify a sentence whether it is an insult or information. This program first annotates related words or phrases in a given sentence; incorporates those annotated elements with the corresponding general semantic structure; then apply some predefined rules for interpreting the basic meaning of the sentence according to that semantic structure and then decides whether it is information or a flame.

Including the introduction in section 1, section 2 describes the related work done in this area; section 3 elaborates the methodology part, which has two **main** sub sections: preprocessing and processing; section 4 contains the description of the tools used in implementation; Results and Discussion are in section 5; limitations of this system is examined in section 6; section 7 outlines the future work; section 8 describes the applications of our system and section 9 has the conclusion.

2. Related Work Done

A flame recognition system is **Smokey**, proposed by Ellen Spertus [1] Smokey looks not only for insulting words in the context in which they are used but also for syntactic constructs that tend to be insulting or condescending. Each sentence is run through a parser and converted into Lisp s-expressions by sed and awk scripts from that parser output. These s-expressions are processed through some semantic rules written in Emacs Lisp, producing a 47-element feature vector based on the syntax and semantics of each sentence. A feature vector for each message is then created by summing up the vectors of each sentence. The resulting feature vectors are evaluated with simple rules, produced by Quinlan's C4.5 decision-tree generator to classify the message as a flame or not. A training set of 720 messages was used by the decision tree generator to determine feature based rules that were able to correctly categorize 64% of the flames and 98% of the non-flames in a separate test set of 460 messages.

2.1 Our contrast with Smokey

- Smokey's semantic rules are some classification rules, which are attempted simultaneously to match some patterns or the syntactical positions of word sequences in a sentence to classify it as a flame or not. But our predefined rules rather tries to extract the semantic information from general semantic structure to interpret the basic meaning of a sentence for distinguishing whether it is a flame or information; not any pattern matching.

- Smokey is message level classification, but our system is sentence level classification.

- We didn't include any sociolinguistic observation or any site-specific information to identify a sentence that not only contains insulting words or phrases but also use them in an insulting manner, as Smokey does. In our system, once insulting words or phrases are found, the semantic information we are getting by only processing the sentence what it gives us, ignoring the surroundings and context.

3. Methodology

3.1 Our annotation scheme in contrast to subjective language

Subjective language is language used to express private states in the context of a text or conversion [2] Researchers from many subareas of Artificial Intelligence and Natural Language Processing have been working on the automatic identification of personal opinions, emotions, sentiments, speculations, evaluations and other private states in language [3] Automatic subjectivity analysis would also be useful to perform flame recognition, email classification etc. [2] Since, flames are viewed as extreme subset of subjective language [4]; we are much more specific and relax. We are considering neither contexts nor surroundings. So, once we find any speech event such as *said, told* etc. we annotate it as a 'factive' event.

Example: *Mary said, "John is an idiot."*

According to subjectivity analysis, including the implicit source <*Writer*> in the above example, here nested sources are

<*Writer, Mary, John*> and it is clearly an opinion at <*Writer, Mary*> level. Thus, 'onlyfactive' property for *Mary's* speaking event *said* is no. And It is an insult at <*Mary, John*> level since insult is 'subset' of subjective language.

In our annotation scheme, root verb *said* is 'factive' at the corresponding dependency structure of **fig-1**, and the complements of this verb form the inner sub-tree rooted at the verb *is,* which is not a 'factive' event and is'insulted'. Since the outer-most root verb *said* is 'factive' and subject *Mary* is a name of a person, the whole sentence is not a flame. Now if we look at the figure from the top: the nested sources will be <*Mary, John*>. Since source <W*riter*> is implicit, we are not considering it.

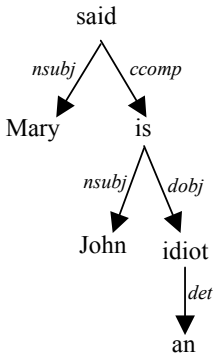


fig -1

3.2 Preprocessing

We need to consider two constraints for meaning interpretation from the parser output:

- It is easier to extract the best semantic information from a dependency structure of a simple sentence rather than a compound or a complex sentence. So, we need to split up a sentence into its corresponding clauses and give each clause a simple construction.

- In a simple sentence, an event or verb must follow its corresponding subject. If this order is reversed, we need to swap.

The steps of our preprocessing part, which will be executed sequentially, are depicted below:

1. Separate each sentence one per line.

2. Replace the factive event 'according to' by 'accorded that' and swap the subject and the event.

For example: *According to Mary, john didn't go downstairs.*
After the operation: *Mary accorded that John didn't go downstairs.*

3. Punctuation marks ("") are used to give a unit scope of speaker's speeches. One or more sentences could be in a scope. Because of punctuation marks are used in a wide variety of ways, only two examples are shown here to express the basic formulation.

First example a paragraph:
"John is waiting for the lift. He didn't go downstairs," Mary replied while talking with Lisa.

After applying some operations, the original sentences in the paragraph will become three separate sentences:
Mary replied, "John is waiting for the lift.
He didn't go downstairs."
Mary replied while talking with Lisa.

If the original sentence was ended just after the word *replied*, then the third sentence will not be present.

Second example: *"John is waiting for the lift," replied Mary, adding, "He didn't go downstairs."*

This sentence will be separated like this:
Mary replied, "John is waiting for the lift."
adding, "He didn't go downstairs."

4. Tag each sentence using stanford-parser (see section 4 for brief description) and store them. Since stanford-parser is a probabilistic parser, give it a full sentence before separate it into clauses.

5. Separate each sentence into clauses by the clause separators. For some separators we need to have some special considerations:

, (comma): We just need to check whether it separates two clauses. If so, then split the sentence, otherwise not. This could be automated.

After separating a sentence, if there is any clause started with a verb, check whether any of the previous clause consist only a nominal subject, then put that nominal subject in front of that verb. For an example: *John Smith, president of the sports club, said, "We will not tolerate it anyway."*

After separating by comma:
John Smith
<,>president of the sports club
<,>said, "We will not tolerate it anyway."

Here, the verb is *said* and the first clause contains only a nominal subject *John Smith.* Put *John Smith* in front of *said.*

Comma just after the speech event will be omitted as shown above.

- (dash), -- (double dash): Consider an example: *We will not tolerate it anyway, because we have to win the match - said John Smith yesterday.*

The above example shows those first and second clauses (separated by 'comma') are the speeches of *John Smith.* The mechanism is, after separating by – (dash), put punctuation marks (") at the beginning of the first clause and at the end of the previous clause of the clause where the speech event (said) found, to put those clauses in a unit scope. Then do the adjustment for punctuation marks as described in step 3.

and: Like comma see whether it separates two clauses or not.

who and *which:* These two separators are considered as same category.

An example: *The speaker here is John Smith, who is also president of the club.*

Since, the noun phrase *John Smith* is at just before the separator *who*, split up a sentence by the separator and put that noun phrase just at beginning of the next separated clause. The above example will be:
The speaker here is John Smith
<who> John Smith is also president of the club.

This process is also same for the separator *<which>*.

Note that, in this preprocessing section, every separator is kept in angle brackets before each separated clause.

The preprocessing tasks in this section, however, are all predetermined. Some preprocessing tasks cannot be predetermined in this section and have to be done at processing part, as described in that section.

3.3 Processing

Before going to the actual processing part, we need to do some preprocessing job each time by manipulating a stack before and after processing of each clause (or a simple sentence). After traversing each dependency tree we are getting some nested sources, which are *agent, experiencer* with their corresponding *events* or verbs. These nested sources and their events are pushed into stack while traversing the tree.

3.3.1 Stack Manipulation

A subject (*agent* or *experiencer*) must exist in the stack with its corresponding event or verb. For all of the figures shown in this paper, stack grows downwards (directed by an arrow); the top of the stack is at the bottom.

Manipulation steps are sequential:
1. Before feeding a clause or a sentence to the parser, we are checking the first word of that clause whether it is a verb. If verb is found, check whether it is a new sentence, or whether this clause was separated by *<while>* or *<because>*. If the

checking returns true then take the last *agent* from the stack not the *experiencer*. Separators *while* and *because*, we call them scope detachers. For any other separators take the *experiencer*.

For example: *Mary said John is an idiot while talking with Lisa.*

After separating by separator <*while*>:
Mary said John is an idiot
<while> talking with Lisa.

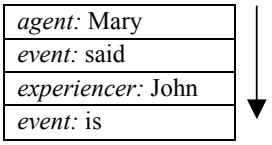


fig-2: stack for the first clause "*Mary said John is an idiot"*

This stack in **fig-2** is constructed by traversing the dependency tree of the first clause of above example, which has a similar dependency tree at **fig-1.**

Now the second clause: *talking with Lisa* starts with a verb *talking* and separated by <*while*>. Then we should take the last agent *Mary*. So, this clause will be: *Mary talking with Lisa.* In case of any other separators, such as <*and*>, the last experiencer *John* should be taken here.

2. Now, detach previous scopes from the stack if it is not empty. A scope can be opened by an *agent* or by an *experiencer*. Detaching a scope means removing a subject (an *agent* or an *experiencer*) with its corresponding *event*.

If the next sentence or clause within a scope of punctuation marks, then detach all the scopes just after the scope opener. For example: *Mary said, "I like fish and vegetables. I hate meat."* Here, the second sentence *I hate meat*, which is in the scope of agent *Mary*. So, detach all of the scopes except the scope opener *Mary.*

If the next sentence is a separated clause but not within punctuation marks, then check the separator and detach scopes reversibly until an *agent* is found or the stack becomes empty. If an *agent* is found and the separator is a 'scope detacher', detach that *agent*.

For example: *Mary told that Lisa said that John is an idiot <and> doesn't know any behavior.*

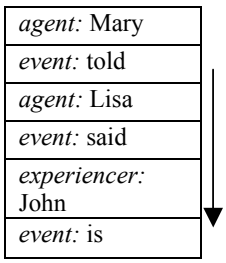


fig-3 (a)

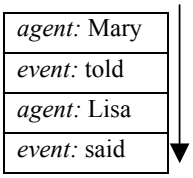


fig-3 (b)

Fig-3 (a) shows the state of the stack after processing the first clause before <*and*> at above example. **Fig-3 (b)** shows that only experiencer *John* has been detached before processing the second clause because <*and*> is not a 'scope detacher'. Otherwise agent *Lisa* will be detached also.

3.3.2 Marking Phase

First make all insulting phrases to one word by putting a '-' between words. Ex: *get a life* will be *get-a-life*. Next mark each word in a sentence if that belongs to any of the following categories. All potential insulting elements are marked with a '*'.

***<phrase>*:** Any insulting phrase such as *get-a-life, get-lost* etc.
***<word>*:** Any insulting word: *stupid, idiot, nonsense, cheat* etc.
***<comparable>*:** If a human being is compared to these objects such as *donkey, dog* etc.
<humanObj>: Any word refers to human being, such as: *he, she, we, they, people, Bangladeshi, Chinese*.
***<attributive>*:** These are the personal attributes of human being such as *behavior, manner, character* etc.

***<factive>*:** All are the speech events such as *said, told, asked* etc. In this context *insults, insulted* are also factive event.
***<evaluative>*:** These verbs are used to evaluate a human being's personal attribute such as *know, show, have, has, expressed* etc.
***<modifier>*:** All modifier verbs: *should, would, must* etc.
***<comparableVerb>*:** These auxiliary verbs are used to compare a human being with the comparable. These are *is, are, was,* and *were*.

Each word will be marked with its 'tag' property. For example the word *behave* will be marked as *<attributive>behave/VB*. We have separate list of lexicon entry for each category described above. The regular expression for matching words or phrases, is case insensitive

3.3.3 Tree Annotation

Now, we have to feed each clause to the parser and the tree is built from the parser output by incorporating those categories at the marking phase as Boolean properties of each node. We also incorporate three basic properties for each node:
label- The word itself
tag- Part of speech of the word
edgeFromParent- Relation between a node and its parent node.

3.3.4 Detection

A set of predefined rules is applied for each node while traversing the tree. While visiting a node we must have two elements:
1. The root node of the current sub-tree, which is being visited.

2. Relation to the root, that means which sub tree we are traversing. Suppose relation *nsubj* indicates that we are traversing the subject part, similarly *dobj* indicates we are traversing object part.

The rules are:
1. When the root verb is 'factive', check whether its subject's 'tag' is NNP (Proper Noun) or PRP (Personal Pronoun) and 'edgeFromParent' property doesn't indicate it is a passive subject, then this subject will become an *agent*. In any other cases it will be an *experiencer.*

Ex: *Peoples say, "We are democratic."*

In this example, *Peoples* is an *experiencer* because its 'tag' is NNS (Common Noun-Plural), although verb *say* is 'factive'.

2. If a dependency structure doesn't contain a verb at the root, and the current node is 'insulted' then set the root to be 'insulted'

Ex: *That nonsense book*

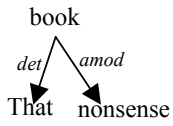


fig –4

The root node is a noun *book*, whose 'insulted' property will be true, since the word *nonsense*, which has its 'insulted' property true, is found as its modifier.

3. If found any insulting word or phrase at the subject part, set the 'insulted' property of the current root to true. The subject will become *experiencer*, no matter what it's corresponding event is (factive or non-factive).

4. If the 'edgeFromParent' property of current 'insulted' node is *dobj* (direct object), or *iobj* (indirect object), *about, with* or *to* then set the parent node (which must be a verb node) to be 'insulted' and its corresponding subject will be an *experiencer,* regardless of the *event* (factive or non-factive).

Ex: *Mary always says that nonsense.*

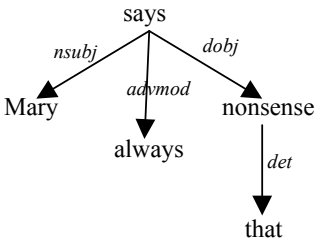


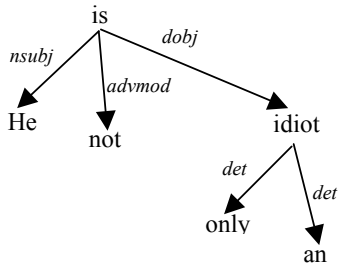
fig-5

In **fig-5**, since the 'insulted' node *nonsense* is direct object of the verb *says,* its 'insulted' property will

be true and subject *Mary* is an *experiencer*, although verb *says* is a 'factive' event.

5. If the root verb has a 'negative' modifier, and the current node has its 'insulted' property true, then check its children. If any of its child nodes has the 'label' *only* then root's 'insulted' property will be true, otherwise false.

Ex: *He is not only an idiot*.



fig–6

At **fig-6** above, the root verb *is* has a negative modifier *not*, and the 'insulted' node *idiot* has a child *only*. So, root's 'insulted' property will be true.

6. If an 'insulted' node's 'edgeFromParent' property is *as, like* or *to* and the subject was 'humanObj' then root will be 'insulted'.

Ex: *He thinks like a donkey.*

In this **fig-7**, node *donkey* is 'comparable' and its 'edgeFromParent' property is *like* and subject *He* was a 'humanObj', so root will be 'insulted'

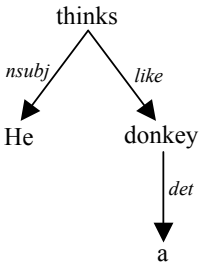


fig-7

7. If current node at subject part is 'comparable' and the root verb is 'comparableVerb', then see whether any 'humanObj' is at object part and set the root's 'insulted' property true. If the 'comparable' node is at object part, then check the subject part for a 'humanObj' and apply the rule.

Ex: *A donkey is what he is.*

In **fig–8**, the node *he* is a 'humanObj' and the subject *donkey*, which has its 'comparable' property true, and the root node *is* also a 'comparableVerb', it is 'insulted'.

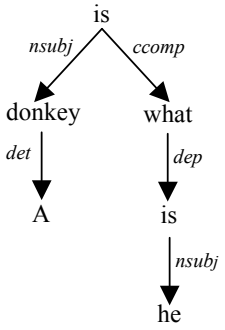


fig-8

8. If a 'humanObj' is a modifier of an 'insulted' node then current root's 'insulted' property will be true. Otherwise, if a 'humanObj' is a modifier of the root verb and root verb also has a 'insulted' node as its modifier then it will be 'insulted'.

Ex-1: *Nobody thinks as an idiot like him.*
Ex-2: *Nobody thinks as an idiot, except him.*

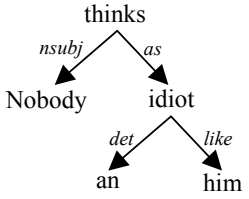


fig- 9 (a)

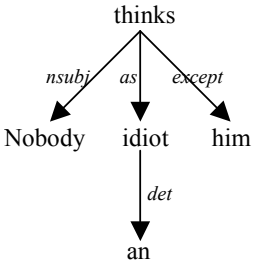


fig- 9 (b)

Fig-9 (a) shows the corresponding dependency structure of Ex-1, where a 'humanObj' *him* is a modifier of an 'insulted' node *idiot,* so root *thinks* will be 'insulted'. For Ex-2, **Fig-9 (b)** shows that 'humanObj' is a modifier of the root verb *thinks* and root also has a modifier *idiot,* then it will be 'insulted' also.

9. If the property of a node is 'attributive' then we got sequentially two checking. First check whether the root node is 'evaluative' or 'comparableVerb'. If that is

true then next checking is whether the root node has a 'negative' modifier or a 'modifier' modifies it. If that is also true then set 'insulted' property of this root to true.

Ex: *John doesn't know any behavior.*

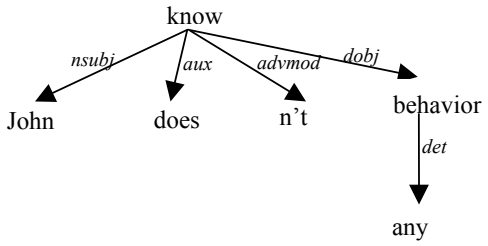


fig-10

In **fig-10** when the node *behavior* will be visited, the 'evaluative' root verb *know* will be 'insulted' since it has a 'negative' modifier *n't* (negative evaluation of someone's personal attribute). Same for the example: *John should know behavior* because the node *know* will be modified by a 'modifier' *should.*

10. If the current node is 'humanObj' and the 'edgeFromParent' property is *by*, then it will be considered as a subject and immediate top subject of the stack has to be changed into current node's label and its subject level can be either *agent* or *experiencer* depending on the condition described in rule no. 1.

Ex: *John was told as an idiot by Mary.*

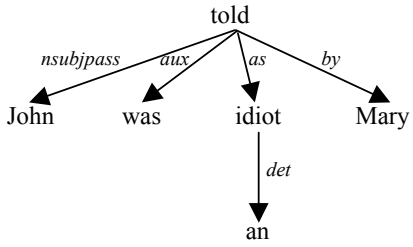


fig -11

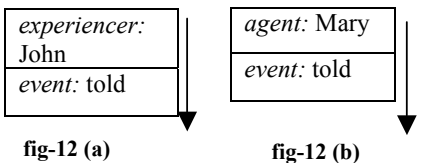


fig-12 (a) **fig-12 (b)**

In **fig-12 (a)** subject *John* was pushed as an *experiencer* into the stack although the root verb is a 'factive' event, because in **fig-11** its 'edgeFromParent' property is *nsubjpass* (passive nominal subject). In **fig-12 (b),** after visiting the node *Mary,* the last subject of the stack is changed to current node's 'label' and its subject level is switched from *experiencer* to *agent*, since the root verb *told* is a 'factive' event and *Mary* is a proper noun (NNP).

Once tree traversing has been completed, following steps are to be executed, if we got the root of a tree has its 'insulted' property true:

1. If currently no scope is open then check the subject at bottom of the stack whether it is an *agent* and its immediate top subject is an *experiencer,* and they are the same person. If they are same, then change that *agent* to an *experiencer*. This step will not be executed if a scope is open.

For example: *Mary said, "Mary is an idiot."*

Here, *Mary* inside the scope of punctuation mark is another *Mary*.

2. Now check whether the stack is empty or bottom of the stack contains an *experiencer*. Then annotate the sentence as an insult.

The processing part described here, is for each clause (or that could be a simple sentence). So, this processing will be repeated for each clause (or for a sentence) until the end of the document.

4. Implementation

We used OpenNLP 1.3.0 for separating sentences within a paragraph. This tool can be accessed online at the following URL: http://aye.comp.nus.edu.sg/portal/RPNLPIR/opennlp_tools_1.3.0,_1.2.0.html. Although, sometimes OpenNLP makes confusions incase of 'dot' or 'full-stop' we are ignoring it, because of its trained feature. For tagging and dependency

parsing we used stanford-parser (version jdk 1.5+), which is available at: http://nlp.stanford.edu/software/lex-parser.shtml. The parser built at Stanford University includes a *typedDependenciesCollapsed* feature for its dependency output format that we are using here.

5. Results and Discussion

This section shows a snapshot of an output of our program. All of the input sentences are taken here arbitrarily. Insulting words or phrases and attributive elements are shown in bold text.

Input Sample Paragraphs:

She said, "Lisa doesn't know any **behavior**." **Get lost** John! You should be punished for your **shameless** work. That **so-called** expert has taken two hours to discuss the problem. Your **ilk** is primarily responsible for most of the ills in this country.

Mary knows that John is **rude**. He should know some **manner**, she replied. He played that shot like a **coward**. According to John, Lisa is so **mean**. He believes that **stupid** Lisa cannot do this. That's why; John was talking about that **stupid** idea in the conference.

Actually, John told that because he usually says that **nonsense**. Lisa said he is an **idiot**. But, that **idiot** said Lisa is a good girl. And she still prays that God heals his heart from all of his **meanness**. Get that socialist out of my pocket!

Output:

[Para: 1 Sentence: 2] Get lost John!
[Para: 1 Sentence: 3] You should be punished for your shameless work.
[Para: 1 Sentence: 4] That so-called expert has taken two hours to discuss the problem.
[Para: 1 Sentence: 5] Your ilk is primarily responsible for most of the ills in this country.
[Para: 2 Sentence: 1] Mary knows that John is rude.
[Para: 2 Sentence: 3] He played that shot like a coward.
[Para: 2 Sentence: 5] He believes that stupid Lisa cannot do this.
[Para: 2 Sentence: 6] That's why; John was talking about that stupid idea in the conference.
[Para: 3 Sentence: 1] Actually, John told that because he usually says that nonsense.
[Para: 3 Sentence: 3] But, that idiot said Lisa is a good girl.
[Para: 3 Sentence: 4] And she still prays that God heals his heart from all of his meanness.

Found: 11 sentences.
Time elapsed: 00 hrs 00 mins 32 secs

Each line of output shows that exactly at which paragraph and at which sentence an insulting content is found. The last two lines show total number of sentences found and time taken to produce the output for the given input in Pentium III 800 MHz~ machine. Since, the program can run in batch mode, time taken for loading the "parser" and "sentence separator" has been excluded. So far, while this paper is being written, each list contains on average 10-11 lexicon entries. Other technical issues (data structures, algorithm, coding style) are also responsible for the time variation. Now, consider the last sentence at the third paragraph of the input, which is clearly an insult but didn't appear at the output. Since it contains neither insulting words, nor phrases according to our lexicon entry. In order to annotate it as a flame, we have to interpret that somebody wants to get a 'humanObj' (socialist) from his/her pocket. This interpretation extremely needs some incorporation of world knowledge for capturing the demeaning of a human being's personal status. Only semantic analysis wouldn't necessarily help.

6. Limitations

1. This system can annotate and distinguish any abusive or insulting sentence only bearing related words or phrases that must exist in the lexicon entry.

2. Our preprocessing part is not yet been full proved to handle all exceptions. For some excessively long or complicated sentences there are possibilities of erroneous output.

3. We didn't yet handle any erroneous input such as misplacing of comma, unmatched punctuation marks etc. at our implemented system.

4. Our performance largely depends on the "Sentence detector of OpenNLP" tools and "stanford-parser". Since stanford-parser is a probabilistic parser, it is not guaranteed that all of its output is right. For those cases, this system also gives the wrong output.

7. Future Work

1. Incorporating world knowledge to annotate a sentence that not only bears insulting words or phrases, but also used as an insulting manner.

2. Not only insults, can be extended to recognize other private states- opinion, emotion, beliefs etc. For example: *Mary thinks that the election was fair.* The verb *thinks* clearly expresses the subject *Mary's* private state at certain **intensity** level according to the **implicit** source *writer*. Here, verb *thinks* is the outer-most root verb of this subjective language and can be evaluated by the corresponding dependency structure.

3. Adding morphological analysis, pragmatics.

4. Adding learning features such as 'supervised learning', this can be based on user feedback.

5. Make it for other languages, such as 'bangla'. In that case we need a 'bangla dependency parser'.

8. Applications

It can be useful for any news site since news mostly represents factual information, or a site that contains informative articles such as 'wikipedia'. Another application could be e-mail filtering because flamers usually send personal attacking messages to individuals via private email.

9. Conclusion

We present a new efficient method for distinguishing flames and information by interpreting the basic meaning of a sentence. However, we are distinguishing flames along with annotating. From psychological point of view flamers usually send abusive messages containing obscene expressions because it affect people most emotionally, if these messages are categorized and restrict a user to send these, human intension to exchange abusive or insulting messages can be significantly reduced.

We describe an elegant approach for extracting the semantic information from the general semantic structure. According to Covington [5] English and all other human languages are "dependency language" and dependency links are closed to the semantic relationships needed for the next approach of interpretation. This paper explores that way of interpretation where each word exhibits its domain specific properties through the word dependency relation in a complete sentence. Simply, this is an introduction of a 'new phase' of domain specific meaning interpretation in a sophisticated method. Moreover, this method can be extendable for annotating personal opinions, beliefs etc., which suggest that the solution is not just an adhoc but has deeper underlying unity.

References:

[1] E. Spertus, "Smokey: Automatic recognition of hostile messages," In Proceedings of the Eighth Annual Conference on Innovative Applications of Artificial Intelligence (IAAI), 1997, pp. 1058-1065.

[2] J. Wiebe, T. Wilson, R. Bruce, M. Bell and M. Martin, "Learning subjective language," Computational Linguistics, Vol. 30, No. 3, 2004, pp. 277-308.

[3] J. Wiebe, T, Wilson, and C. Cardie, "Annotating expressions of opinions and emotions in language," Language Resources and Evaluation, Vol. 39, Issue 2-3, 2005, pp. 165-210.

[4] M. Martin, "Annotating flames in Usenet newsgroups: a corpus study," For NSF Minority Institution Infrastructure Grant Site Visit to NMSU CS department, 2002.

[5] M. A. Covington. "A fundamental algorithm for dependency parsing," Proceedings of the 39th Annual ACM Southeast Conference, 2001, pp. 93-95