WordSleuth: Deducing Social Connotations from Syntactic Clues

Shannon Stanton

Honors Thesis 2011

WordSleuth: Deducing Social Connotations from Syntactic Clues

Shannon Stanton
University of California, Irvine
Information and Computer Science
Campus-wide Honors Program
sstanton@uci.edu
shannonnstanton@gmail.com

0. Abstract

The realm of social and emotional connotation is often thought to be the purview of humans rather than machines. Namely, humans are generally capable of recognizing social connotations including emotions (such as embarrassment), intentions (deception and persuading), attitudes (confidence and disbelief), and tone (formality, politeness, rudeness), and recent work has suggested that machines may also be capable of this feat (Pearl and Steyvers 2010). This study extends the work done by Pearl and Steyvers, improving the data gathering methodology, feature extraction, and machine learning classification. Prior to the WordSleuth project, a major barrier to researching social cues transmitted through text has been a lack of annotated data. WordSleuth, an online Game-With-a-Purpose (von Ahn 2006), solves this problem, creating an effective means of encouraging a wide variety of participants to generate and annotate data. Salient linguistic features can then be extracted from the data gathered and used to train and test machine learning algorithms, effectively teaching machines to identify social connotations in text. In particular, as machines still currently lag behind human capabilities, this study extends Pearl and Steyvers' work by examining more complex linguistic features and exploring more sophisticated machine learning methods, with the aim of substantially improving machine recognition of social connotation.

1. Introduction

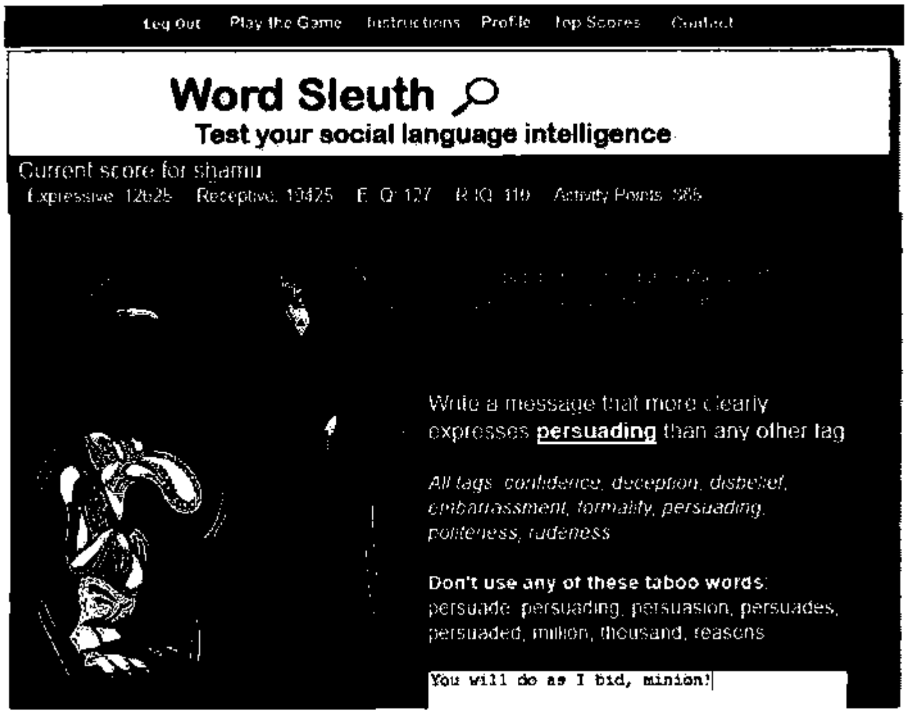
An important question in computational linguistics research is how non-linguistic information, such as emotions, intentions, attitudes, and tone, can be derived from language text. People are generally capable of it, but so far, machines have lagged significantly behind human capability. One approach is to identify possible features humans use, such as low level syntactic cues, and extract them from the input, allowing machine learning algorithms to make use of them, potentially even better than humans. This research project focuses on low level syntactic clues present in plain text.

The primary technical barrier to research in social connotation up until this project was a lack of socially annotated data. In order to extract such social information from text, we must first have a reference point constituted by sufficient examples of each category: a database of reliable messages reflecting human perceptions of both the intended and perceived social information. We therefore cannot simply automate the process (until after this project), since the machine learning itself requires training data to learn from. We need also a diversity of examples and styles to generalize from, so simply annotating existing works may be insufficient, and is, at the very least, extremely time-consuming. While some sources of information annotated for select specific categories exist, such as a database for deception created from the online game Mafia Wars (Zhou and Sung 2008), these sources do not reflect the breadth of social connotations we are looking for. Thus, the goal is to obtain messages generated and annotated by many people. Simple survey techniques can only bring in so much data due to limited scope and appeal. Pearl and Steyvers' (2010) solution to this problem: a game, specifically, a game-with-a-purpose (von Ahn 2006), that can automate the acquisition of data and increase the amount provided by volunteers by making participation more enjoyable (Pearl and Steyvers 2010). We call that game WordSleuth.

2. Creating WordSleuth

2.1 The function and purpose of the WordSleuth game

WordSleuth's game play is bimodal, facilitating the gathering of both new annotated messages and annotations of old messages. In the first mode, message generation, players are presented with a contextual picture for inspiration and one of eight social cues, and asked to create a message that expresses that cue more than any of the others, without using particular taboo words that might make the task of identification too easy. This mode enables the generation of new annotated data, but that alone would be insufficient: we also want to gather data about people's perceptions of the message's social category.



Word Sleuth

Test your social language intelligence

Current score for shamu

Expressive: 12625 Receptive: 13425 E.Q: 127 R.Q: 119 Activity Points: 565

Write a message that more clearly expresses **persuading** than any other tag

All tags: confidence, deception, disbelief, embarrassment, formality, persuading, politeness, rudeness

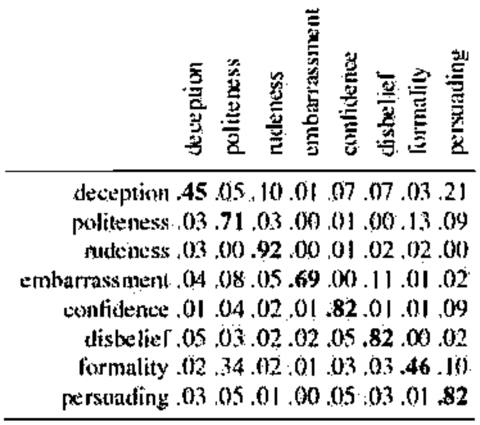
Don't use any of these taboo words: persuade, persuading, persuasion, persuades, persuaded, million, thousand, reasons

You will do as I bid, minion!

Illustration 0: *Message Creation Mode*

In the second mode, cue identification, players are presented with a message and the contextual image used to generate it, and asked to identify which of the eight social cues the message best communicates. This mode allows users to peer review each others' submissions, providing information about whether messages identified represent "good" examples of their social cue. Ideally, messages that are the best examples of their category are always agreed upon, while the worst examples show a high degree of confusion among the guessers. It also increases the appeal of the game play, as it appears players have a strong preference to the relatively simpler task of identification, perhaps because it is faster and less cognitively taxing, providing more instant gratification.

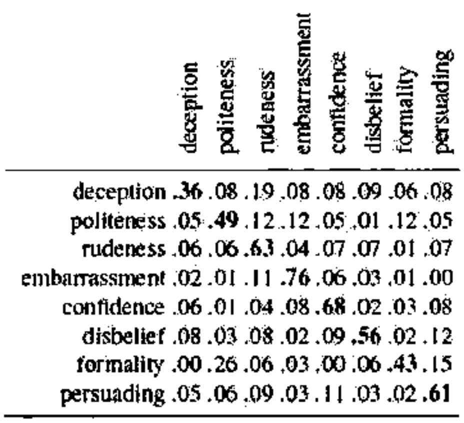
It has been shown that this type of communal effort of non-experts is capable of producing data as reliable as that generated by few experts (von Ahn 2006). For convenience and ease of comparison, the following tables show the initial results obtained by Pearl and Steyvers' participants, when the database included 1176 messages and 3198 annotations. The reliability of the data increased dramatically when we considered messages that have been agreed upon for at least 50% of at least two annotations (Pearl and Steyvers 2010).



	deception	politeness	rudeness	embarrassment	confidence	disbelief	formality	persuading
deception	**.45**	.05	.10	.01	.07	.07	.03	.21
politeness	.03	**.71**	.03	.00	.01	.00	.13	.09
rudeness	.03	.00	**.92**	.00	.01	.02	.02	.00
embarrassment	.04	.08	.05	**.69**	.00	.11	.01	.02
confidence	.01	.04	.02	.01	**.82**	.01	.01	.09
disbelief	.05	.03	.02	.02	.05	**.82**	.00	.02
formality	.02	.34	.02	.01	.03	.03	**.46**	.10
persuading	.03	.05	.01	.00	.05	.03	.01	**.82**

Table 3: Confusion matrix for the human participants, where the majority of participants agreed on a message's intended social information and at least two participants labeled the message. The rows represent the intended social information for a message while the columns represent the labeled social information, averaged over messages and participants.

(Pearl and Steyvers 2010)



	deception	politeness	rudeness	embarrassment	confidence	disbelief	formality	persuading
deception	**.36**	.08	.19	.08	.08	.09	.06	.08
politeness	.05	**.49**	.12	.12	.05	.01	.12	.05
rudeness	.06	.06	**.63**	.04	.07	.07	.01	.07
embarrassment	.02	.01	.11	**.76**	.06	.03	.01	.00
confidence	.06	.01	.04	.08	**.68**	.02	.03	.08
disbelief	.08	.03	.08	.02	.09	**.56**	.02	.12
formality	.00	.26	.06	.03	.00	.06	**.43**	.15
persuading	.05	.06	.09	.03	.11	.03	.02	**.61**

Table 4: Confusion matrix for the machine learning classifier. The rows represent the intended social information for a message while the columns represent the labeled social information.

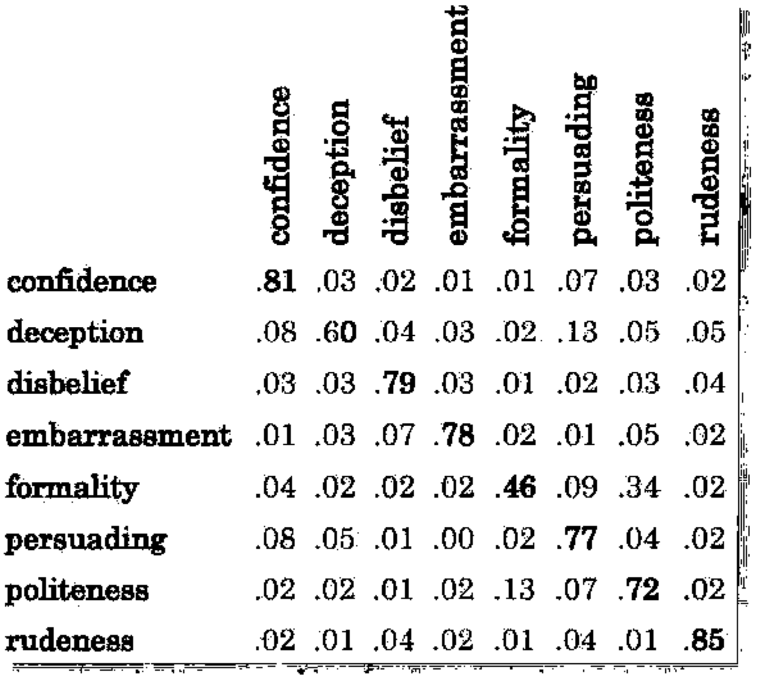
(Pearl and Steyvers 2010)

WordSleuth was originally created as an offline game, which limited its effectiveness in reaching participants and gathering data. A much larger database is required to truly generalize about such a nebulous subject as social connotations.

2.2 Bringing WordSleuth online

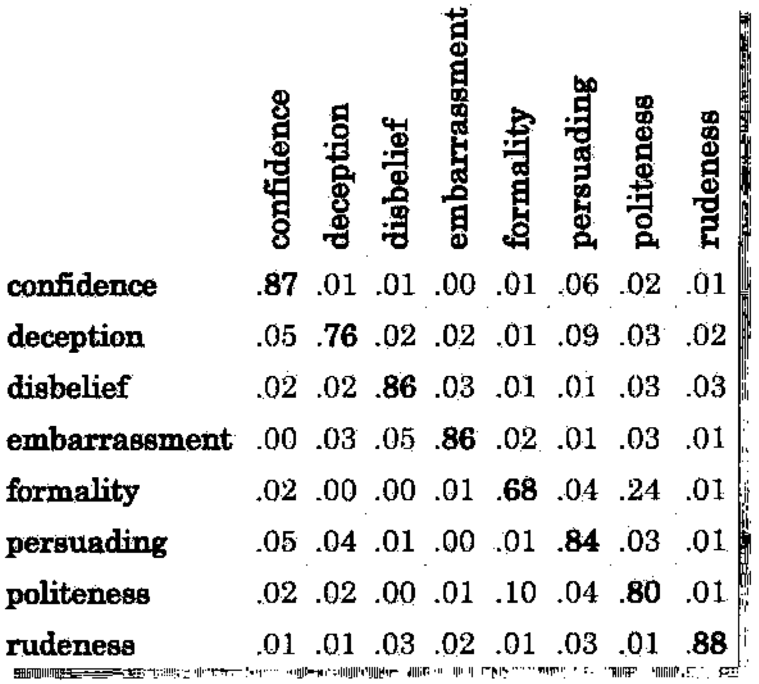
A solution to the data deficit problem is putting the game online (see http://gwap.ss.uci.edu/ for the current instantiation), increasing its accessibility to the general public and increasing the amount of data generated. HTML templates were used for the webpages forming the front end of the system, driven by Perl CGI scripts. More modern, flashy methods such as Ruby-on-Rails were contemplated and discarded in favor of quick prototyping and known compatibility with popular browsers. Finally, the front-end system was integrated with a mySQL database, an improvement in efficiency, availability, and methodology from the text files previously used.

Results to date are promising. Since bringing the game online in January 2011, the number of annotations has increased dramatically while the number of messages created has nearly doubled. As of May 2011 the database contains just over 3,500 messages and 20,000 annotations.



	confidence	**deception**	**disbelief**	**embarrassment**	**formality**	**persuading**	**politeness**	**rudeness**
confidence	**.81**	.03	.02	.01	.01	.07	.03	.02
deception	.08	**.60**	.04	.03	.02	.13	.05	.05
disbelief	.03	.03	**.79**	.03	.01	.02	.03	.04
embarrassment	.01	.03	.07	**.78**	.02	.01	.05	.02
formality	.04	.02	.02	.02	**.46**	.09	.34	.02
persuading	.08	.05	.01	.00	.02	**.77**	.04	.02
politeness	.02	.02	.01	.02	.13	.07	**.72**	.02
rudeness	.02	.01	.04	.02	.01	.04	.01	**.85**

***Table** 0: Human annotations for database (as of May 2011)*
Mean accuracy: 0.74



	confidence	deception	disbelief	embarrassment	formality	persuading	politeness	rudeness
confidence	**.87**	.01	.01	.00	.01	.06	.02	.01
deception	.05	**.76**	.02	.02	.01	.09	.03	.02
disbelief	.02	.02	**.86**	.03	.01	.01	.03	.03
embarrassment	.00	.03	.05	**.86**	.02	.01	.03	.01
formality	.02	.00	.00	.01	**.68**	.04	.24	.01
persuading	.05	.04	.01	.00	.01	**.84**	.03	.01
politeness	.02	.02	.00	.01	.10	.04	**.80**	.01
rudeness	.01	.01	.03	.02	.01	.03	.01	**.88**

Table 1: *Human annotations for reliable messages (as of May 2011)*
Mean accuracy: 0.84

In general, the expansion of the database has seen an increase in user accuracy in identifying the intended social cue, as well as the reduction of certain ambiguities. Confusion of deception for confidence, for example, has been halved, even without filtering for reliably annotated messages. Rudeness is still easiest for users to identify, but by a slimmer margin. However, some sources of confusion remain prominent, for example formality for politeness, and less so, the reverse.

2.3 Improvement Feature: Taboo word list

One potential complication that may arise with gathering data in a competitive framework is the possibility of amassing messages that are artificially representative of their classifications. Players motivated by point gain may specifically craft messages that are trivial to guess by including the social tag in the message or using words that are too closely related to the tag. For example, the task of identifying "politeness" in a message is trivialized if every message assigned to that category has the word "please". Therefore, users should be prevented from using select words. Rejecting messages containing variations of the tag and the tag itself was a simple starting point and solved the first half of the problem, but we also needed some way of tracking words that were becoming over represented in the database. Our solution was to dynamically generate a list of taboo words based on the theory of mutual information.

Mutual information is a measure of the inter-dependence of two variables (Peng 2005): in this case, word frequency and social category. Two independent variables should have a mutual information score of 0, while two variables that are dependent and closely related will have a higher score than two non-closely related. The following equation was used,

$$\mathit{Mutual\ Information}(x,y)=\log\frac{(p(x:y))}{(p(x)*p(y))}$$

where,

$$p(x:y)=\mathit{probability\ of\ word\ x\ given\ category\ y}\ ,$$
$$p(x)=\mathit{probability\ of\ word\ x\ among\ all\ words}\ ,$$
$$p(y)=\mathit{probability\ of\ category\ y\ among\ all\ categories}\ .$$

For each social category, the words with the highest mutual information score are declared to be taboo in the game, and players are not allowed to use them when generating a message for that particular category. Common words, such as articles and pronouns, should be automatically excluded, since they are evenly distributed among all the categories.

Taboo list functionality was implemented with a Perl script to calculate the mutual information scores for each word in each social category in the current database, set to update approximately once per day. Thus the taboo lists are dynamically updated to reflect the state of the database, automatically without requiring the direct supervision of the researchers. The following code fragment illustrates the implementation of the mutual information calculation:

```
	# calculate p(x) = # occurences of word/#total words
	my $px = $wordFrequency{$word}/$totalWords;
	#calcuIuate p(y) = #occurences of a given tag/totalMessages
	my $py = $tagCount{$category}/$totalMessages;
	#calculate p(x|y) = #word x in tag y/#words in tag y
	if (! exists $wordCount{$category} || $px ==0) #if $py is 0,
 bigger problems to be alerted to (ie. social tag not existing)
	{
		$pointWiseMutualInfo = 0;
	}
	else
	{
		my $pxGy = $count/$wordCount{$category};
		$pointWiseMutualInfo = log($pxGy/$px/$py); #log(p(x|y)/
 (p(x)*p(y)))
	}
	$mutualInfo{$category}{$word} = $pointWiseMutualInfo;
```

Code_Fragment 1: *tabooListGenerator.pl: calculating mutual information*

For example, as of May 2011 each category yielded the following taboo words:

Category	Taboo List: Top 7
confidence	**wil, modest, mvp, talkies, rule, scruffles, sorts**
deception	**recommend, spreadsheet, dastardly, issue, nerdy, jan, suntan**
disbelief	**beats, megaphone, guitar, twenty, vat, goatse, smoothly**

embarrassment	stew, conscious, mins, grease, mighty, private, spade
formality	delivery, abuse, form, grammy, greetings, martin, distinguished
persuading	million, thousand, reasons, captain, poverty, carrots, tonic
politeness	nicely, grateful, bumping, rough, shore, orphans, scores
rudeness	kangaroo, facts, uncalled, scum, listed, spotty, gingers

Table 2: *Taboo list results (as of May 2011)*

Many of these words are intuitively related to their given category: "modest" in confidence, "recommend" in deception, "million", "thousand", "reasons" for persuading, etc. However, many appear at first glance to be out of place.

A useful, if unexpected, outcome of applying this methodology was the identification of words that were non-intuitively highly correlated with particular categories. For example, just after the game went online in January 2011, the taboo list generator yielded "nancy" for confidence. Yet "nancy" does not seem to be a word that one would intuitively associate with the social category confidence; it seems rather arbitrary. In fact, that unigram was an artifact of the message generation system. In the beginning, when the game was offline and the database relatively small, a user happened to use the name "Nancy" in several messages for the category confidence. Because there were so few repeated words in general and that one happened to be used enough in a particular category, it had a relatively high mutual information score, even though it may not be truly representative of the category. Making "nancy" taboo for the confidence category prevents users from creating additional instances correlating the unigram to the category, thus eventually lowering its mutual information score. Thus, taboo functionality reduces the effect of coincidental correlation.

Eventually, as the database grows, trends can be examined to set an appropriate absolute boundary on the mutual information score, rather than simply using the highest relative scores. The taboo list should eventually resemble the game for which it was named and represent words that are highly correlated for each category within the current database. It is important to note that this will not necessarily reflect the correlation present in general language usage, since this model actively discourages high correlations. Therefore, taboo list functionality increases both the depth and breadth of data represented by discouraging trivially obvious words such as the categories themselves and by dynamically identifying and reducing coincidentally high correlations of words to categories.

3. Using WordSleuth

The data gathered in the WordSleuth database cannot be simply directly fed to a computer and expect coherent results. It must first be parsed and processed for salient, numerable features. Furthermore, many feature are only present for a few messages, listing only those features present for each message reduces the dimensionality of the data set, thus increasing the efficiency of the algorithms.

3.1 Features

Originally, Pearl and Steyvers used 12 features extracted for each message: the number of word types, number of word tokens, ratio of types to tokens, number of punctuation marks, number of question marks, number of exclamation marks, number of main clauses, average characters per word, mean log frequency of words used, and lists of unigrams, bigrams, and trigrams that appear more than once in the data set. This project added the following features: number of interrobangs, ratio of exclamation to question mark, average words per main clause, number of sub-clauses, average words per sub-clause, and accuracy and precision scores for human performance on each message. For example, interrobangs appear in the disbelief category more often than others, while formality and deception are often expressed with numerous sub-clauses distancing the speaker from the audience. Accuracy and precision scores give a sense of the usefulness of a particular message as an exemplar. Accuracy is calculated as the percentage of times a particular message was correctly identified, while precision represents a measure of the agreement (or lack of confusion) of the guessers, calculated as a percentage of the maximum possible entropy. Maximum entropy (which is 3 bits for an 8 category choice) represents the state of maximum confusion (each category is guessed 1/8 of the time), and thus the lowest precision (0). Minimum entropy (0) represents complete certainty (such as when all guessers guess the same category) and thus the highest degree of precision (1.0 or 100%). Thus precision is calculated:

$$precision=\frac{(H_{max}-H_x)}{H_{max}}$$

where,

$$H_x=\sum p(x)*\log_2\frac{1}{(p(x))}$$

and,

$$H_{max}=H(\frac{1}{8},\frac{1}{8},\frac{1}{8},\frac{1}{8},\frac{1}{8},\frac{1}{8},\frac{1}{8},\frac{1}{8})=3$$

I considered several ways to calculate precision, such that precision should represent the amount of agreement of the guessers on a particular message.

First, I considered precision to be simply the frequency of the most common guess, but quickly realized some flaws with this hypothesis. This calculated precision could never be lower than accuracy, and yet it occurs in other domains that precision is lower than accuracy. Further, this metric would not be sufficiently fine-grained. For example, consider 2 messages, one that is guessed 50% one category and 50% another, to be represented (.5, .5) for short, and the other, that is guessed 50% one category, 25% another, and 25% a third (.5, .25, .25). In both cases, this calculation for precision would yield .5, but it seems intuitively that the second case represents a higher degree of confusion among the participants, since more categories were under consideration.

Next, I considered various ways of penalizing precision based on the number of categories guessed. However, this method is insufficiently fine-grained as well. Consider 2 messages, the first (.5, .25, .25) and the second (.5, .24, .01). Simply accounting for the most commonly guessed and the number of categories would calculate the same precision for each of these messages, but again intuition says the second one might represent a lower degree of confusion, since the third category has so few guessers compared to the other two. Precision should take into account the relative frequency of each category guessed as well.

The entropy ratio calculation solves these problems. It is possible for a message to have lower precision than accuracy (such as, for example, (.3, .1, .1, .1, .1, .1, .1, .1)), and there is sufficiently high granularity to distinguish the aforementioned cases.

3.2 Algorithms

Preliminary research with the machine learning algorithm Sparse Multinomial Logistic Regression (Pearl and Steyvers 2010) showed performance nearly on par with human performance, but not quite. Just as there is variation among the performance of individual humans on learning tasks, different machine learning algorithms vary in performance, with their own sets of strengths and weaknesses. This paper examines additional algorithms in an attempt to reach human proficiency.

3.2.1 KNN: K-Nearest-Neighbors

3.2.1.1 KNN Background

As a "peer pressure" multinomial classification algorithm, K-Nearest-Neighbors operates on an inductive principal of classifying a test data point based on the training data points proximate to it. Each unknown data point adopts the classification of those closest to it, or, in the case of disagreement, the most common classification of nearby training points. Let there be two subsets of data, one for training whose classifications are known to the algorithm and one for testing whose classifications are unknown to the algorithm, but known to the evaluator of algorithms. (Here the "correct classification" is defined as that specified by the user when the message was generated.) For each data point in the test data, KNN calculates the Euclidean distance between that data point and each data point in the training subset. It then assigns the classification of the test data point to the most common classification of the K training cases with the smallest distances.

There is some concern about efficiency. For n test cases and d training cases, the algorithm runs in at minimum O(n*d) time and can do no better, making it inefficient for large values of n or d. In reality, because of the way we parse features, n depends on both the number of messages and the number of features parsed, and thus grows rather quickly. KNN may not be practical if the database continues to grow in size as hoped.

To begin with, KNN was run on the database toward the end of May 2011 and fed only the features originally extracted by Pearl and Steyvers in 2010. Next, KNN was applied to the additional low level features. In both cases, performance was averaged over values of N ranging from 1 to 55.

3.2.1.2 KNN Results

	confidence	deception	disbelief	embarrassment	formality	persuading	politeness	rudeness
confidence	**.80**	.04	.01	.02	.02	.06	.02	.03
deception	.09	**.55**	.03	.01	.00	.11	.01	.00
disbelief	.03	.02	**.79**	.02	.03	.04	.04	.03
embarrassment	.02	.06	.03	**.79**	.02	.02	.03	.02
formality	.02	.01	.01	.02	**.60**	.03	.31	.00
persuading	.07	.06	.02	.01	.03	**.76**	.03	.02
politeness	.03	.02	.03	.11	.02	.05	**.71**	.02
rudeness	.02	.01	.07	.03	.00	.06	.01	**.80**

Table 3: *KNN on May 2011 data, original features*
Mean accuracy 0.76

	confidence	deception	disbelief	embarrassment	formality	persuading	politeness	rudeness
confidence	**.17**	.14	.12	.10	.08	.12	.16	.10
deception	.13	**.13**	.16	.09	.12	.16	.09	.12
disbelief	.12	.10	**.13**	.12	.14	.16	.12	.10
embarrassment	.06	.16	.11	**.11**	.16	.14	.15	.11
formality	.10	.15	.14	.18	**.10**	.10	.15	.07
persuading	.13	.13	.16	.11	.13	**.08**	.15	.12
politeness	.15	.08	.16	.08	.15	.16	**.18**	.13
rudeness	.12	.09	.12	.11	.11	.09	.15	**.20**

Table 4: *KNN on May 2011 data, all features*
Mean accuracy 0.24

Notably, KNN's mean performance on the original features is equivalent to human performance on all messages. Surprisingly, KNN performed much worse with all features than with the original features alone. However, KNN is sensitive to dimensionality and proximity, and it may be that the new features confused the algorithm by creating the illusion of proximity.

KNN is a naïve algorithm in that it overlooks certain patterns apparent in the data, such as clustering. Furthermore, as an inductive algorithm, it is only able to learn from the training set. Thus, it is unable to make use of test cases themselves, which would be particularly beneficial when the differing categories are highly interspersed, as is the case here. Transductive clustering suffers neither of these deficiencies.

3.2.2 Transductive Clustering

The primary difference between induction and transduction in this case is the ability to make use of information from unlabeled points in the test subset (Chapelle, Scholkopf, and Zien 2006). While inductive KNN would only use training data near a test point, transduction also considers other as yet unlabeled test points and is able to make use of their proximity once labeled. Furthermore, clustering is able to take advantage of the patterns that exist in the data beyond the first level of nearby points.

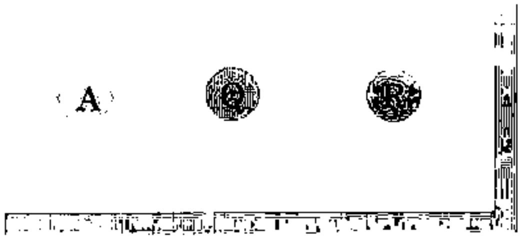


***Illustration 1:** Training point A (gold hexagon), test points Q and R*

For example, consider Illustration 1: if training point A is near test point Q, and Q is near test point R, transductive clustering is able to infer that A and Q and R should have the same label, since they form a cluster, because the unlabeled test point Q between A and R joins them together. Both KNN and transductive clustering would label both Q and R with category gold hexagon, but with differing underlying logic. Inductive KNN would label Q according to A (gold hexagon), and then R according to A (also gold hexagon), and not explicitly understand that Q and R are the same category, because it is blind to point R when considering point Q (and vice-versa). This difference in logic becomes more salient if there is an additional training point of a different category, as follows.

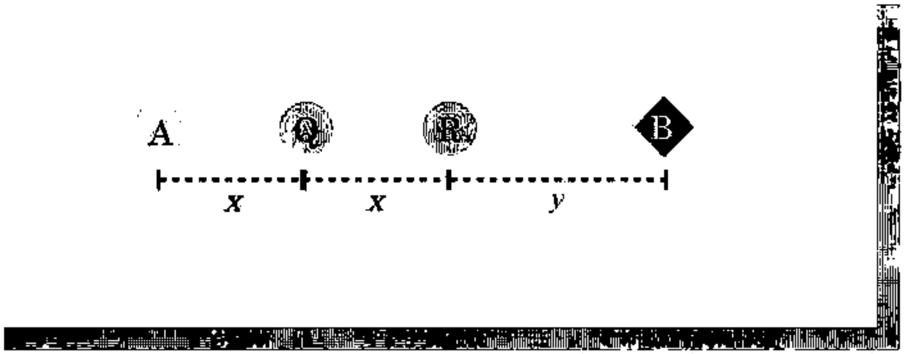


***Illustration 2:** Training points A (gold hexagon) and B (blue diamond), and test points Q and R. Distances* x *and* y *such that* x < y < 2x.

Now consider Illustration 2, in which another training point B exists (labeled with category blue diamond), closer to R than A is to R, and of a different label than A. KNN (K=1) would label R according to B, rather than according to A, since R is nearer to B, though intuitively A and R should probably belong in the same cluster, and thus the same category label. This intuition grows stronger with the introduction of more unlabeled points, as shown in Illustration 3.

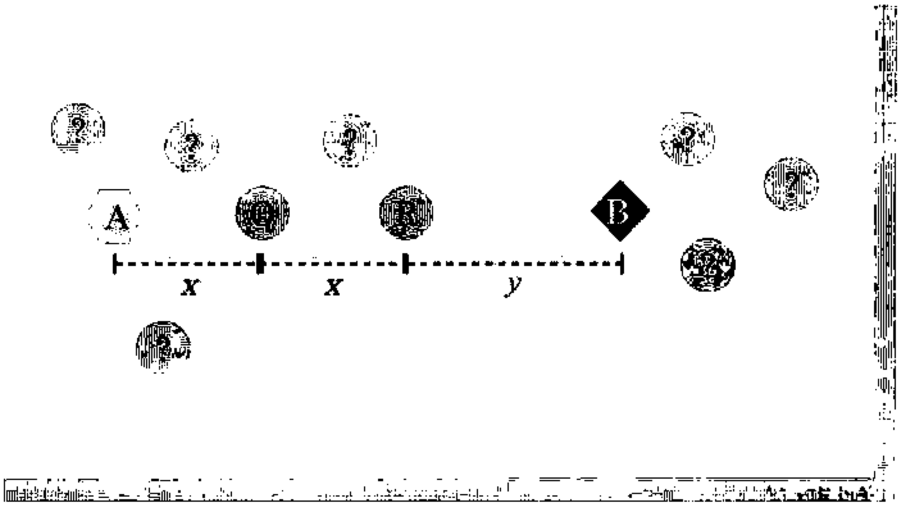


***Illustration 3:** Additional unlabeled data points enhance the intuition of two clusters, where the left cluster should be gold hexagon, and the right cluster blue diamond.*

3.2.2.1 Transductive Agglomerative Clustering

Transductive Agglomerative Clustering works by merging nearby points into clusters (Gashler 2011). Once a labeled point is merged into a cluster, the entire cluster gains the label of that point, and thus do all the unlabeled points within the cluster. In theory this sounds plausible. However, the mean accuracy of this algorithm was only about 0.13 (below the baseline of 0.15), when tested with 10 repetitions of 10-fold cross-validation. Upon closer examination of the algorithm, one finds that clusters of differing labels are never joined, which bodes ill for data that shows many small, interspersed clusters, or clusters that have some conflicting labels. These are in fact the characteristics inherent to the current WordSleuth data set.

3.2.2.2 Transductive Graph Cutting

Transductive Graph Cutting uses a min-cut/max-flow algorithm to separate out the various labels present in the data and deliminate clusters accordingly (Gashler 2011). When run on the May 2011 data set with only the original features present with both 10 repetitions of 10-fold cross-validation and 10 repetitions of 2-fold cross-validation, the mean accuracy was 0.97, much higher than the other algorithms or human annotations. A result so high seemed to indicate the potential of overfitting; to truly determine, additional testing data is required, but running 2 fold cross-validation to reduce the ratio of training to test data suggests the results are robust. When run on the same data set and cross-validation, but with all features extracted, the mean accuracy was 0.98, showing that the additional features did not cause this algorithm the level of confusion as inductive KNN experienced.

4. Future Directions

With additional time, the WordSleuth project could benefit from further research done in several areas, including additional feature research and machine learning techniques. For example, this paper only examines relatively low level syntactic clues; the success of certain classifiers relative to humans on such low level cues suggests that humans may cue into these low level clues, but they probably also use higher level data, including sentence structure. Input messages could be parsed into syntax trees to examine high level syntactic structures. I began tentative work on approximating these structures with simple parts of speech tagging which shows promise, but time constraints did not permit. Additional machine learning algorithms not examined in this paper, including additional inductive and transductive algorithms would be interesting to look into, and combining the strengths of multiple algorithms with methods such as bagging could yield more powerful, consistent, and robust results.

5. Works Cited

Gashler, Mike. *Waffles.* http://waffles.sourceforge.net/docs.html. March 2011.

Pearl, L. & Steyvers, M. (2010). Identifying Emotions, Intentions, & Attitudes in Text Using a Game with a Purpose. Proceedings of NAACL-HLT 2010 Workshop on Computational Approaches to Analysis and Generation of Emotion in Text. Los Angeles, CA: NAACL.

Peng, H.C., Long, F., and Ding, C., "Feature selection based on mutual information: criteria of max-dependency, max-relevance, and min-redundancy," IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 27, No. 8, pp. 1226-1238, 2005.

von Ahn, L. 2006. Games With A Purpose. IEEE Computer Magazine, June 2006: 96-98.

Zhou, L., Burgoon, J., Nunamaker, J., and Twitchell, D. 2004. Automating linguistics based cues for detecting deception in text-based asynchronous computer mediated communication. Group Decision and Negotiation, 13: 81-106.

Zhou, L. and Sung, Y. 2008. Cues to deception in online Chinese groups. Proceedings of the 41st Annual Hawaii international Conference on System Sciences, 146. Washington, DC: IEEE Computer Society.

6. Appendix Contents: code written specifically for WordSleuth

6.1 Taboo list generation script: taboo_list_generator.pl

6.2 Feature extraction script: get_features_shamu.pl

6. Appendix: WordSleuth Code

6.1 Taboo list generation script: taboo_list_generator.pl

```
#!/usr/bin/perl
use strict;

# this script should be passed the
following arguments:
# 1. the name of the input file
#   hint: The input file needs to be
formatted such on each line, the tag
#         comes first, separated by
*** then the message, then a new
line.
#         And do make sure the
messages don't contain the delimiter.
# 2. the number of taboo words to get

my %tabooLists = &main($ARGV[0],
$ARGV[1]); #wrap main
print "Result: \n";

while (my($k, $v) =
each(%tabooLists))
{
    print ("$k: ",join(",
",@$v),"\n");
}

sub main
{
my $numArgs = $#ARGV + 1;

if ($#ARGV+1 != 2) # must have
exactly 2 args
{
    print "Please specify the proper
arguments next time\n";
    print "You should specify the
name of the input file and the number
of taboo words per category\n";
    exit;
}

if ($ARGV[1] < 0) # check validity of
second arg
{
    print "Invalid arg 2, please try
again\n";
    exit;
}

foreach my $argnum (0 .. $#ARGV)
{
   print "$ARGV[$argnum]\n";
}
```

```
open(inputFile, $ARGV[0]);
my %chart; #category tag => hash of
word to frequency
my %wordCount; #number of total words
in a given category tag
my %wordFrequency; #total times the
word appears overall all tags
my $totalWords = 0;
my $totalMessages = 0;
my %tagCount;

while (<inputFile>)
{
    my($line) = $_; # store local $_
in temp
    chomp($line); # strip line of
trailing newline

    # parse line into social tag and
message
    # which are deliminated by ***

    print "\n$line\n\n";

    #my($tag, $message) = ($line
=~ /^(.*)\s+\*\*\*\s+(.*)$/);
    $line =~ /^(.*)\s+\*\*\*\s+(.*)
$/;
    my $tag = $1;
    my $message = $2;

    print "tag: 6 $tag 9\n";

    if (exists $tagCount{$tag})
    {
      $tagCount{$tag} +=1;
    }
    else
    {
     $tagCount{$tag} = 1;
    }
    $totalMessages +=1;

    #print "message: 6 $message\n";
#for debugging

    $message =~ s/'/'/g; #convert all
mystery ticks to apostrophes
    #$message =~ s/'//g; #remove all
apostrophes¡

    #print "message: $message\n";
#for debugging

    $message =~ s/[^'\w]/ /g;
#replace all punctuation besides
apostrophes/underscores with white
space
    $message = lc($message);
    #print "message: $message\n";
#for debugging

    my @words = split(/\s+/,
$message); #deliminate on one or more
white spaces

    #print "words: "; #for debugging
    #print join(':', @words); #for
debugging
    #print "\n"; #for debugging

    foreach (@words)
    {
     my $word = $_;
     #print " my word! $word: \n";
#for debugging
     next if ($word eq '' || $word eq
'\''); #ignore these non-words

     $totalWords +=1;
     #print "has $totalWords
words!\n"; #for debugging

     if
(exists($wordFrequency{$word}))
     {
         $wordFrequency{$word} +=1;
     }
     else
     {
         $wordFrequency{$word} = 1;
     }

     if (exists($wordCount{$tag}))
     {
         #print "old tag"; #for
debugging
         $wordCount{$tag} +=1;
         #print "  $wordCount{$tag}
"; #for debugging
     }
     else
     {
         #print "new tag"; #for
debugging
         $wordCount{$tag} = 1;
     }


     if (exists($chart{$tag}{$word}))
     {
         #print "charting old word";
#for debugging
         $chart{$tag}{$word} +=1;
     }
     else
     {
```

```
         #print "charting new word";
#for debugging
         $chart{$tag}{$word} = 1;
         #print "$chart{$tag}{$_}";
#for debugging
     }
    }
}

# print hashes for clarity, or
comment out if desired
print "\nwordCount: \n"; #for
debugging
while ( my($k, $v) =
each(%wordCount))
{
    print "$k -> $v\n";
}

print "\nwordFrequency: \n";
while ( my($k, $v) =
each(%wordFrequency))
{
    print "$k -> $v\n";
}

while ( my($k, $v) = each(%chart))
{
    print "\n$k: \n";
    while ( my($l, $u) = each(%$v))
    {
     print "$l -> $u, ";
    }
}

my %mutualInfo; #category =>

while (my($category,$v) =
each(%chart))
{
    while (my($word,$count) = each(%
$v))
    {
     my $pointWiseMutualInfo = 0;
     if ($totalWords == 0)
     {
         print "No words found,
bye\n";
         exit;
     }
     if ($totalMessages ==0)
     {
         print "No messages found,
bye\n";
         exit;
     }
     # calculate p(x) = # occurences
of word/#total words
     my $px = $wordFrequency{$word}/
```

```
$totalWords;
      #calculuate p(y) = #occurences
of a given tag/totalMessages
      my $py = $tagCount{$category}/
$totalMessages;
      #calculate p(x|y) = #word x in
tag y/#words in tag y
      if (! exists
$wordCount{$category} || $px ==0) #if
$py is 0, bigger problems to be
alerted to (ie. social tag not
existing)
      {
          $pointWiseMutualInfo = 0;
      }
      else
      {
          my $pxGy = $count/
$wordCount{$category};
          $pointWiseMutualInfo =
log($pxGy/$px/$py); #log(p(x|y)/
(p(x)*p(y)))
      }
      $mutualInfo{$category}{$word} =
$pointWiseMutualInfo;
    }
}

my %tabooLists;
print "Final Results:\n";
while (my($key,$val) =
each(%mutualInfo))
{
    my $tempKey = $key;
    my %temp1 = %{$mutualInfo{$key}};
    my @temp = sort {$temp1{$b} <=>
$temp1{$a}} keys %temp1;
    $tabooLists{$key} = ();
    print "arg1: $ARGV[1]\n";
    for (my $i = 0; $i < $ARGV[1];
$i++)
    {
      print "temp[$i]: $temp[$i]\n";
      push(@{$tabooLists{$key}},
$temp[$i]);
    }

    print "$key: \n";
    while (my($k1,$v1) = each(%$val))
    {
      print "$k1 -> $v1, ";
    }
    print "\n";
}

print "total words: $totalWords\n";
print "total messages:
$totalMessages\n";
```

```
return %tabooLists;
} # end subroutine main
```

6.2 Feature extraction script: get_features_shamu.pl

```
#!/usr/bin/perl
use switch;

# usage:
# get_features_shamu.pl -createdinput
$createdfilename -guessedinput
$humanfilename -outputbase
$outputfilebasename

# Modified by Shannon Stanton for
parsing the current database format
# Requires 2 input files:
human_guesses and created_items (in
tab deliminated format)
# Can be fetched from database at
http://madlab.ss.uci.edu/pma/index.ph
p?db=gwap
#
# human_guesses:
# guess_id message_id time_stamp
guesser session correct_social_tag
guessed_social_tag guessed_correctly
#
# created_items:
# message_id message time_stamp
creator difficulty session_id
set_social_tag picture_file
times_guessed times_guessed_correctly
flags

# Some notes on style:
# Generally: Tend toward explicit,
verbose code.  This is for research,
#   and that research is not about
Perl subtleties, and
#   future researchers needn't spend
hours on Perl subtleties.
# Ampersands: As I understand it,
Perl 5 no longer requires & preceding
function
#   calls.  However, since they
(generally) improve syntax
highlighting
#   and point out that a (user
defined) function is being called,
I'm
#   keeping them in the code.
Apologies for inadvertant
inconsistency.
# Parameters to subroutine calls:
&foo; and &foo(); are in fact
different.
# References: I don't like them.  I
avoid using them in this script.
# Underscores: Are not currently
consistent.
# :? Conditionals: Seem to behave
unexpectedly when combined with
increments
#   (+= and ++).  Beware.
# Strict: Not compatible with use
strict; so don't!

# for extracting features from
messages
# assumes input takes the form of an
excel spreadsheet dumped to a txt
file
# for example:
#
#Alias    Timestamp  Social Cue
     Interaction     SessionID
     AliasR     MessageID  Message
     Guess Correct     PictureFile
#LisaEx    time1 deception  generate
     "32532787"      "1"  Oh sure -
we're just here for some fresh air,
see the sites, that kind of thing.
We have absolutely no intention of
making a mess in your nice pond,
nope.  We would never ever do
something like that.   Spick and
span, that's us.
     20451652.png
#LisaEx    time10     embarrassment
     generate   "32532787"      "10"
     Holy crap, I had no idea that
you were the Green Trio…please go
ahead.  I can't believe I didn't
recognize you…must be my low blood
sugar, the heat, I'm so sorry…go
right ahead.
     20451652.png
#labsubject18   4/30/09 11:24
     persuading generate   "596770"
     "1026239"  If you take care of
all four kids, I'll buy you the new
mattress that you wanted!
     20819897.png
#
# all entries are separated by tabs



# The script produces several
separate output files.

# The first
($outputfilebasename.messageinfo) has
the following format
#
#
MessageID\tMessageContent\tSocialGoal
Intended\tGenerator\tFeature1\tFeatur
```

```
e2...\tFeaturen
# 596770\tIf you take care of all
four kids, I'll buy you the new
mattress that you
wanted!\tpersuading\t...
#
# with the following features
included
#
# (1) how often guessed right
(requires counts of correct guesses
for message and total guesses for
message)
# (1a-1h) how often guessed as
particular socialCues (deception,
politeness, rudeness, embarrassment,
#  confidence, disbelief, formality,
persuading


# The second output file
($outputfilebasename.featurelist) has
the following 2-column format
#
#
$feature_idnum\t$feature_description
# 1\tWordTokens
# 2\tWordTypes
# ...
# 343| word: forgot
# ...
# 3043| bigram: forgot my
# ...
# 30043 | trigram: forgot my shoes

# Current features extracted:
#
# (2) word types in message (unique
words in message)
# (3) word tokens in message (total
words in message)
# (4) type to token ratio (use type
and token counts to calculate)
# (5) # of punctuation marks in
message (can include ellipsis)
# (5a) # of questions marks in
message
# (5b) # of exclamation marks in
message
# (6) # of separate
sentences/questions in message (main
clauses)
# (7) average word length per message
# (8) mean log frequency of words
used (compared against words used in
all messages)
# (9) through (n) count of vocabulary
item used (doesn't include words only
used once)
```

```
# (b1) through (bn) count of bigrams
used (doesn't include bigrams only
used once)
# (t1) through (tn) count of trigrams
used (doesn't include trigrams only
used once)

# The third output file
($outputfilebasename.messagefeatures)
has the following 3-column sparse
data format
#
# $message_idnum\t$feature_idnum|
t$feature_value
# 108898\t343\t2
#
# Note: only non-zero values are
listed (this is what makes it a
sparse data format)

# The fourth output file is the
.userinfo file and includes
# (1) the name of the user
# (2) the total number of messages
generated
# (3) the percent of messages
generated that were correctly guessed
(expressor %)
# (4) the total number of messages
guessed
# (5) the percent of messages
correctly guessed (sleuth %)


# Design decisions: for considering a
message created "correctly" we might
# want to look at the number of
correct guesses associated

{
    $debugging = 1; #1 is true, 0 is
false, mark false if you don't want
to print all the obnoxious helpful
debug lines

    &process_options();

    my $outputFileName =
$opt_outputbase."\.debuggy";
    open(DEBUG, ">$outputFileName")
|| die("Couldn't open debugging file
$outputFileName\n");

    &initialize_globals();

    #process each of the 2 input
files to put all the relevant raw
data in hashes %allMessages and
%allUsers
```

```
    &process_created();
    &process_guesses();

    &extractFeatures();

    &writeOutputFiles(); #only to be
done after filling the raw data
hashes

    if ($debugging)
{ &print_hashes();}
    close(DEBUG);
}

sub process_options{
    use Getopt::Long;
  &GetOptions("createdinput=s",
"guessedinput=s", "outputbase=s",
#createdinput and guessedinput and
output are required
            "filter:s",
"printheader:s"); # optional header
printing (default is 'yes', can be
set to 'no') and filter (as in filter
for reliable messages)
}

# expects the raw data from input
files to be encapsulated in the
hashes %allUsers and %allMessages
# The second output file
($outputfilebasename.featurelist) has
the following 2-column format
#
#
$feature_idnum\t$feature_description
# 1\tWordTokens
# 2\tWordTypes
# ...
# 343| word: forgot
# ...
# 3043| bigram: forgot my
# ...
# 30043 | trigram: forgot my shoes

  # current features extracted (does
not reflect order, order is
determined alphabetically by
description):
  # (2) word types in message (unique
words in message)
  # (3) word tokens in message (total
words in message)
  # (4) type to token ratio (use type
and token counts to calculate)
  # (5) # of punctuation marks in
message (can include ellipsis)
  # (5a) # of questions marks in
message
  # (5b) # of exclamation marks in
message
  # (5c) # of elipses (...) in
message
  # (6) # of separate
sentences/questions in message
  # (7) average word length per
message
  # (8) mean log frequency of words
used (compared against words used in
all messages)
  # (9) accuracy of guesses (correct
guesses/total guesses)
  # (10) precision of guesses (see
calculation)
  # (11) through (n) count of
vocabulary item used (doesn't include
words only used once)
  # (b1) through (bn) count of
bigrams used (doesn't include bigrams
only used once)
  # (t1) through (tn) count of
trigrams used (doesn't include
trigrams only used once)
 # rest for part of speech info

# extractFeatures:
# Input: None.
# Output: None.
# Effects: Updates globals
%directFeaturesNew and
%directFeaturesOld for every
#   unique message id in %allMessages
# Expects: %allMessages should be
filled correctly prior to calling
this method.
sub extractFeatures()
{
    print("...extracting
features\n");
    print(DEBUG "Feature
Extraction:\n");
    # remeber, 2 feature hashes for
old and new features
    # %directFeaturesOld and
%directFeaturesNew

    print(DEBUG "---1st loop----\n");
    foreach my $id
(sort(keys(%allMessages)))
    {
      #my %messageWords = ();
      my $numWords = 0;
      my $numLetters = 0;
      my $message = $allMessages{$id}
{"message"};

      #@messageWords =
split(/\s|\.|\?|\!|â€¦|\,|\"|\
(|\)|;/, $message); #shamu note:
semicolon not used in original
     my @messageWords =
&get_word_list($message);
     my %messageWordsHash = ();

     foreach my $word (@messageWords)
{
         if ($word =~ /\w/) #if it
has any word characters in it
         {
           $numWords++;
# insert features pertaining to upper
case here!!!
           $word =~ tr/[A-Z]/[a-z]/;
#shifts everything to lower case
           if
(exists($messageWordsHash{$word})){

$messageWordsHash{$word}++; #
increment
           }
           else{

$messageWordsHash{$word} = 1; #
initialize
           }
           # update %allWords
           if
(exists($allWords{$word})){
                $allWords{$word}++;
           }else{
                $allWords{$word} = 1;
           }

           #calculate number of
letters in the word
           my @letters = split(//,
$word);
           foreach $letter (@letters)
{
                if($letter =~ /\w/){
                  $numLetters++;
                } # end if
           } # end foreach $letter
         } # end if ($word =~ /\w/)
     } # end foreach word

     #&update_allUnigrams($id);
     &update_allBigrams($id);
     &update_allTrigrams($id);

     #calculate number of word types
in message (unique words)
     my $wordTypes =
scalar(keys(%messageWordsHash)); #
the number of word types, not the
types themselves
     $directFeaturesOld{$id}
```

```
{"wordTypes"} = $wordTypes;
     $directFeaturesOld{$id}
{"wordTokens"} = $numWords;
     $directFeaturesOld{$id}
{"typesToTokensRatio"} = $wordTypes/
$numWords; #should be less than or
equal to 1

     # punctuation features time
     my $punctCount=0;
     while($message
=~ /\.|\?|\!|\,|-|;/g){$punctCount+
+;}

     #my @punctCount =
split(/\.|\?|\!|\'|\â€¦|\,|;/,
$message); #ok, the funny symbol \â€¦
seems to be an artifact of operating
systems and text editors conversions:
usually it seems to stand in for
apostrophes (single quote, not the
back tick)
     # shamu note: the split method
does not seem to work correctly,
particularly in that it does not
count matches at the end of a string
     # shamu note: I have added
semicolon here, which was not done in
the original version
     # my $#numPunct = $punctCount;#
+ 1; # $#array list gives the index
of the last element, so yes a pound
symbol that is not a comment mark

     $directFeaturesOld{$id}
{"punctMarks"} = $punctCount;
     # question marks?
     #my @qmCount = split(/\?/,
$message);
     #my $numQM = $#qmCount;
     my $numQM =
&get_num_qm($message);
     #while($message =~ /\?/g)
{$numQM++;}

     #if($message =~ /\?$/){ #if the
message ends in a question mark, add
one more
     #    $numQM = $numQM + 1;
     #}
     $directFeaturesOld{$id}
{"questionMarks"} = $numQM;

     # excalamation marks!!!
     #my @emCount = split(/\!/,
$message);
     #my $numEM = $#emCount;
     #if ($message =~ /\!$/){
     #    $numEM = $numEM + 1;
```

```
    #}
    my $numEM =
&get_num_em($message);
    #while($message =~ /\!/g)
{$numEM++;}
    $directFeaturesOld{$id}
{"exclamMarks"} = $numEM;

    # new: interrobangs!? ?! (seem
to have a high correlation with
disbelief, depending on the ratio of
interro to bang)
    my $numIB =
&get_num_ib($message);
    #while($message =~ /\!\?|\?\!/)
{$numIB++;} #bad code, infinite loop
    $directFeaturesNew{$id}
{"interrobangs"} = $numIB;

    # new: ratio of question marks
to exclamation marks
    my $QMtoEM =
&get_qm_to_em($message);
    $directFeaturesNew{$id}
{"QMtoEMRatio"} = $QMtoEM;

    # new: elipses:
    $directFeaturesNew{$id}
{"elipses"} =
&get_num_elipses($message);

    # new: length of longest elipses
run
    $directFeaturesNew{$id}
{"elipsesRun"} =
&get_longest_elipses_run($message);

    # calculate and add in the
number of main clauses, as delimited
by . ? and ! and ; (shamu note:
semicolon was not used in the
original version

    # the split method should work
this time
    my @mcCount =
split(/\.|\?|\!|;/, $message);
    my $mcNum = 0; # not just the
size of the split, since we might
have repititious punctation: ie.
don't count !!!!! as four clauses
    # count how many contain words
    foreach my $partOfMC (@mcCount)
{
       if ($partOfMC =~ /\w/) { #o
goody it contains wordy things, let's
count them
        $mcNum++;
       }
```

```
    }
    $directFeaturesOld{$id}
{"mainClauses"} = $mcNum; #number of
main clauses
    $directFeaturesNew{$id}
{"mainClausesAv"} = $numWords/$mcNum;
#average number of words per main
clause #new!
    # subclauses time! as delimited
by , : ( ) / " and /- / dash-space
and / -/ space-dash

    $directFeaturesNew{$id}
{"accuracy"} =
&calculateAccuracy($id);
    $directFeaturesNew{$id}
{"precision"} =
&calculatePrecision($id);

   } # end for every message id in
allMessages foreach my $id
(sort(keys(%allMessages))) (first)

   # Hashes: %allWords, %allBigrams,
%allTrigrams should be fully updated
   &update_allFeatures(); # and now
@allFeaturesList should reflect the
grams

   print(DEBUG "second loop\n");
   # now that we've counted all the
unigrams, bigrams, and trigrams,
enter them into the feature list
   # extractFeatures: bigrams
   foreach my $id
(sort(keys(%allMessages)))
   {
    print(DEBUG "id: $id\n");
      my @messageWords =
&get_word_list($allMessages{$id}
{"message"});

    # unigrams: single words:
    for my $unigram
(sort(@messageWords))
    {
       if ($allWords{$unigram} > 1)
# only count if it occurs more than
once in the entire input
       {
        $directFeaturesOld{$id}
{"word:".$unigram} =
$allWords{$unigram};
       }
    }

    # bigrams: 2 words
    my %bigrams =
&get_bigram_list(@messageWords);
```

```
     for my $bigram
(sort(keys( %bigrams )))
     {
         if ($allBigrams{$bigram} >
1){ #only count as a feature if it
occurs more than once in the entire
input
          $directFeaturesOld{$id}
{"bigram:".$bigram} =
$bigrams{$bigram};
         }
     }

     # trigrams:
     my %trigrams =
&get_trigram_list(@messageWords);
     for my $trigram
(sort(keys(%trigrams)))
     {
         if ($allTrigrams{$trigram}
>1){
          $directFeaturesOld{$id}
{"trigram:".$trigram} =
$trigrams{$trigram};
         }
     }

    } # end foreach my $id
(sort(keys(%allMessages))) (second)

} # end sub extractFeatures

sub writeOutputFiles #expects no
arguments
{
    &writeMessageInfoFile();
    &writeFeatureListFile();
    &writeMessageFeaturesFile();
    #&writeUserInfoFile(); #I'm just
going to veto this one, since I have
no use for the file anyway
    &writearffFile();
    &writeConfusionMatrix(); #for
human guesses
    &writeReliableMatrix(); # for
human guesses
}

# Inputs: A hash! (Not a hash
reference)
# Reliable is defined as having at
least 50% accuracy and more than 2
votes
sub getReliable
{
    my %hash = @_;
    my %answer = ();

    for my $key(keys(%hash))
```

```
    {
     if( ($hash{$key}{"totalGuesses"}
> 1) && $directFeaturesNew{$key}
{"accuracy"} >= .5)
     {
         %{$answer{$key}} = %
{$hash{$key}};
     }
    }

    return %answer;
}

sub writeReliableMatrix
{
    my $outputFileName =
$opt_outputbase."_reliable\.confusion
matrix";
    open(OUT, ">$outputFileName") ||
die("Couldn't open reliable confusion
matrix file $outputFileName\n");

    print("...writing $outputFileName
file\n");


    my %confusionMatrix = ();
    #intialize matrix with a row and
a column for each social cue and a
row total for each row

    my $totalAcc = 0;
    my $total = 0;

    foreach my $targetRow
(@socialCues)
    {
     foreach my $guessCol
(@socialCues)
     {
         $confusionMatrix{$targetRow}
{$guessCol} = 0;
     }
     $confusionMatrix{$targetRow}
{"rowTotal"} = 0; #just as long as
"rowTotal" is never a social cue
which would be wierd o,O
    }
    my %allMessagesReliable =
&getReliable(%allMessages);

    # count data from allMessages
that are reliable!
    foreach my $id (keys
(%allMessagesReliable))
    {
     $totalAcc +=
$allMessagesReliable{$id}
{"totalCorrectGuesses"};
```

```
      $total +=
$allMessagesReliable{$id}
{"totalGuesses"};

      my $target =
$allMessagesReliable{$id}
{"targetCue"};
      foreach $guessCue (@socialCues)
#add guesses for this message
      {
          my $x =
$allMessagesReliable{$id}{$guessCue};
          $confusionMatrix{$target}
{$guessCue} += $x;
          $confusionMatrix{$target}
{"rowTotal"} += $x;
      }
    }

    print(OUT join("\t",
@socialCues)."\n"); #header

    # divide each cell by row total
    foreach my $targetRow
(sort(@socialCues))
    {
      my @row = (); my $i = 0;
      foreach my $guessCol
(sort(@socialCues))
      {
          if
($confusionMatrix{$targetRow}
{"rowTotal"} != 0)

{   $confusionMatrix{$targetRow}
{$guessCol} /=
$confusionMatrix{$targetRow}
{"rowTotal"};}
          $row[$i] = "($targetRow,
$guessCol):".
$confusionMatrix{$targetRow}
{$guessCol};
          $i++;
      }
      print(OUT join("\t",
@row)."\n\n");
    }

    my $meanAcc=0;

    if ($total != 0)
    {
      $meanAcc=$totalAcc/$total;
    }

    print(OUT "mean accuracy:
$meanAcc\n");
```

```
    close(OUT);
}

sub writeConfusionMatrix # expects no
arguments and that allMessages has
been properly filled in
# target cue accross the rows
# guess cue down the colums
# divide cells by row total
# rows should sum to one (columns may
not)
{
    print("...writing
$opt_outputbase.confusionmatrix
file\n");
    my $outputFileName =
$opt_outputbase."\.confusionmatrix";
    open(OUT, ">$outputFileName") ||
die("Couldn't open confusion matrix
output file $outputFileName\n");

    my %confusionMatrix = ();

    my $totalAcc = 0;
    my $total = 0;

    #intialize matrix with a row and
a column for each social cue and a
row total for each row
    foreach my $targetRow
(@socialCues)
    {
      foreach my $guessCol
(@socialCues)
      {
          $confusionMatrix{$targetRow}
{$guessCol} = 0;
      }
      $confusionMatrix{$targetRow}
{"rowTotal"} = 0; #just as long as
"rowTotal" is never a social cue
which would be wierd o,O
    }

    # count data from allMessages
    foreach my $id (keys
(%allMessages))
    {
      $totalAcc += $allMessages{$id}
{"totalCorrectGuesses"};
      $total += $allMessages{$id}
{"totalGuesses"};

      my $target = $allMessages{$id}
{"targetCue"};
      foreach $guessCue (@socialCues)
#add guesses for this message
      {
          my $x = $allMessages{$id}
```

```
{$guessCue};
         $confusionMatrix{$target}
{$guessCue} += $x;
         $confusionMatrix{$target}
{"rowTotal"} += $x;
      }
    }

    print(OUT join("\t",
@socialCues)."\n"); #header

    # divide each cell by row total
    foreach my $targetRow
(sort(@socialCues))
    {
     my @row = (); my $i = 0;
     foreach my $guessCol
(sort(@socialCues))
     {
         if
($confusionMatrix{$targetRow}
{"rowTotal"} != 0)

{   $confusionMatrix{$targetRow}
{$guessCol} /=
$confusionMatrix{$targetRow}
{"rowTotal"};}
         $row[$i] = "($targetRow,
$guessCol):".
$confusionMatrix{$targetRow}
{$guessCol};
         $i++;
     }
     print(OUT join("\t",
@row)."\n\n");
    }


    my $meanAcc=0;

    if ($total != 0)
    {
     $meanAcc=$totalAcc/$total;
    }

    print(OUT "mean accuracy:
$meanAcc\n");

    close(OUT);
}

# tab deliminated
# format:
#MessageID\tMessageContent\tSocialGoa
lIntended\tGenerator\taccuracy\tpreci
sion\tguessedConfidence\tguessedDecep
tion...\tguessedRudeness
# where accuracy = percent guessed
correctly = (times guessed
correctly / times guessed)
# where precision = max(times guessed
tag x / times guessed) for each tag
# 596770\tIf you take care of all
four kids, I'll buy you the new
mattress that you
wanted!\tpersuading\tLisaEx\t.5\t...
sub writeMessageInfoFile()
{
    print("...writing
$opt_outputbase.messageinfo file\n");
    my $outputFileName =
$opt_outputbase."\.messageinfo";
    open(OUT, ">$outputFileName") ||
die("Couldn't open
$outputFileName\n");

    # print header information
    unless($opt_printheader eq "no"){
     print(OUT
"MessageID\tMessageContent\tSocialCue
Generated\tGenerator\taccuracy\tpreci
sion\t");
     print(OUT
"Guess:confidence\tGuess:deception\tG
uess:disbelief\t");
     print(OUT
"Guess:embarrassment\tGuess:formality
\tGuess:persuading\t");
     print(OUT
"Guess:politeness\tGuess:rudeness\n")
;
    }

    foreach my $messageID (keys
(%allMessages))
    {
     my $messageContent, $targetCue,
$creator, $accuracy, $precision;
     $messageContent =
$allMessages{$messageID}{"message"};
     $targetCue =
$allMessages{$messageID}
{"targetCue"};
     $creator =
$allMessages{$messageID}{"creator"};
     if ($allMessages{$messageID}
{"totalGuesses"} == 0) # no guesses
for this message
     {
         $accuracy = 0;
         $precision = 0;
     }
     else
     {
         $accuracy =
$allMessages{$messageID}
{"totalCorrectGuesses"}/
$allMessages{$messageID}
```

```
{"totalGuesses"};
        $precision =
&calculatePrecision($messageID);
     }
     print(OUT
"$messageID\t$messageContent\t$target
Cue\t$creator\t$accuracy\t$precision"
);
     print(OUT
"\t$allMessages{$messageID}
{\"confidence\"}\t$allMessages{$messa
geID}
{\"deception\"}\t$allMessages{$messag
eID}
{\"disbelief\"}\t$allMessages{$messag
eID}
{\"embarrassment\"}\t$allMessages{$me
ssageID}
{\"formality\"}\t$allMessages{$messag
eID}
{\"persuading\"}\t$allMessages{$messa
geID}
{\"politeness\"}\t$allMessages{$messa
geID}{\"rudeness\"}\n");

    }
    close(OUT);
}

sub calculateAccuracy
{
    my $messageID = shift;
    if($allMessages{$messageID}
{"totalGuesses"} == 0) #div by 0
error
    {
     return 0;
    }
    return ($allMessages{$messageID}
{"totalCorrectGuesses"}/
$allMessages{$messageID}
{"totalGuesses"});
}

sub calculatePrecision #expects a
valid $messageID and that
%allMessages has been properly filled
and that the global list @socialCues
is correct
{
    my $messageID = shift;

    if (!exists
$allMessages{$messageID}) #bad
    {
     print(DEBUG "Error! $messageID
not a valid messageID\n");
     return 0; #just so we don't
crash
```

```
    }

    if ($allMessages{$messageID}
{"totalGuesses"} == 0)
    {
     print(DEBUG "Warning! $messageID
has been guessed 0 times\n");
     return 0; #just to prevent
crashing the script
    }
    # entropy: sum{p(x)*log2[1/p(x)]
    # note: perl's log is natural by
default (log base e) so divide by
log(2) to get log base two
    my $numCat = scalar(@socialCues);
#number of social categories
    my $maxEntropy =
log($numCat)/log(2); # = 3 for 8
categories

    my @pX = ();
    my $entropy = 0;

    foreach my $cue (@socialCues)
    {
     my $px =
$allMessages{$messageID}{$cue}/
$allMessages{$messageID}
{"totalGuesses"};
     if ($px != 0) {$entropy +=
($px)*(log(1/$px)/log(2));}
     if ($debugging) {print(DEBUG
"cue:$cue px:$px\n");}
    }

    my $precision = ($maxEntropy-
$entropy)/$maxEntropy;
    if ($debugging) {
     print(DEBUG "entropy of
$messageID is $entropy\n");
     print(DEBUG "precision of
$messageID is $precision\n");}
    return $precision;
}

# format: FeatureID \t  Feature Label
# (without the spaces: for clarities
sake only)
# deliminated by \t (tab)
sub writeFeatureListFile #expects no
args
{
    print("...writing
$opt_outputbase.featurelist file\n");
    my $outputFileName =
$opt_outputbase."\.featurelist";
    open(OUT, ">$outputFileName") ||
die("Couldn't open
$outputFileName\n");
```

```
    print(OUT "Feature ID:\tFeature
Label:\n");

    #my $id = 1; # IKR: because
matlab starts indexing at 1
    for my $key
(sort(keys(%allFeatures)))
    {
      print(OUT
"$allFeatures{$key}\t$key\n");
    }

    close(OUT);
}

# format: MessageID \t FeatureID \t
Value (all numeric)
# note: featureID starts indexing at
1
sub writeMessageFeaturesFile #expects
no args
{
    print("...writing
$opt_outputbase.messagefeatures
file\n");
    my $outputFileName =
$opt_outputbase."\.messagefeatures";
    open(OUT, ">$outputFileName") ||
die("Couldn't open
$outputFileName\n");

    print(OUT
"MessageID\tFeatureID\tValue\n");

    for my $messageID
(sort(keys(%allMessages)))
    {
      for my $featureLabel(sort(keys(%
{$directFeaturesOld{$messageID}})))
      {
          my $value =
$directFeaturesOld{$messageID}
{$featureLabel};
          if ($value) #don't print the
0 values (sparsity)
          {
            print(OUT
"$messageID\t$allFeatures{$featureLab
el}\t$value\n");
          }
      }

      for my $featureLabel(sort(keys(%
{$directFeaturesNew{$messageID}})))
      {
          my $value =
$directFeaturesNew{$messageID}
{$featureLabel};
          if ($value) #don't print the
0 values (sparsity)
          {
            print(OUT
"$messageID\t$allFeatures{$featureLab
el}\t$value\n");
          }
      }
    }
    close(OUT);
}

# lower priority
sub writeUserInfoFile #expects no
args
{
    print("...writing
$opt_outputbase.userinfo file\n");
    my $outputFileName =
$opt_outputbase."\.userinfo";
    open(OUT, ">$outputFileName") ||
die("Couldn't open
$outputFileName\n");

    close(OUT);
}

# format:
# @RELATION file
# @ATTRIBUTE MessageID NUMERIC
# @ATTRIBUTE FeatureID NUMERIC
# @ATTRIBUTE FeatureValue NUMERIC
# @ATTRIBUTE class {deception,
persuading, confidence, formality,
politeness, rudeness, embarrassment,
disbelief}
# @DATA
# #,#,#,string
#
# where data entries are comma
deliminated and rows separated by \n

sub writearffFile #expects no args
{
    print("...writing
$opt_outputbase.arff file\n");
    my $outputFileName =
$opt_outputbase."\.arff";
    open(OUT, ">$outputFileName") ||
die("Couldn't open
$outputFileName\n");

    print(OUT '%comment!'."\n");
    print(OUT '@RELATION '.
$opt_outputbase."\n");
    print(OUT '@ATTRIBUTE MessageID
NUMERIC'."\n");
    print(OUT '@ATTRIBUTE FeatureID
NUMERIC'."\n");
    print(OUT '@ATTRIBUTE
FeatureValue NUMERIC'."\n");
    print(OUT '@ATTRIBUTE class {');
    print(OUT "$socialCues[0]");
    for (my $i=1; $i<=$#socialCues;
$i++)
    {
     print(OUT ", $socialCues[$i]");
    }
    print(OUT "}\n");
    print(OUT '@DATA'."\n");

    for my $messageID
(sort(keys(%allMessages))) #MARK
    {
     my $targetCue =
$allMessages{$messageID}
{"targetCue"};
     for my $featureLabel(sort(keys(%
{$directFeaturesOld{$messageID}})))
     {
         my $value =
$directFeaturesOld{$messageID}
{$featureLabel};
         if ($value) #don't print 0
values (sparsity)
         {
          print(OUT "$messageID,
$allFeatures{$featureLabel},$value,
$targetCue\n");
         }
     }

     for my $featureLabel(sort(keys(%
{$directFeaturesNew{$messageID}})))
     {
         my $value =
$directFeaturesNew{$messageID}
{$featureLabel};
         if ($value) #don't print 0
values (sparsity)
         {
          print(OUT "$messageID,
$allFeatures{$featureLabel},$value,
$targetCue\n");
         }
     }
    }

    close(OUT);

    my $outputFileName =
$opt_outputbase."_original"."\.arff";
    print("...writing $outputFileName
for original features only\n");

    open(OUT, ">$outputFileName") ||
die("Couldn't open
$outputFileName\n");
    print(OUT '%comment! This uses
only the original features'."\n");
    print(OUT '@RELATION '.
$opt_outputbase."\n");
    print(OUT '@ATTRIBUTE MessageID
NUMERIC'."\n");
    print(OUT '@ATTRIBUTE FeatureID
NUMERIC'."\n");
    print(OUT '@ATTRIBUTE
FeatureValue NUMERIC'."\n");
    print(OUT '@ATTRIBUTE class {');
    print(OUT "$socialCues[0]");
    for (my $i=1; $i<=$#socialCues;
$i++)
    {
     print(OUT ", $socialCues[$i]");
    }
    print(OUT "}\n");
    print(OUT '@DATA'."\n");

    for my $messageID
(sort(keys(%allMessages))) #MARK
    {
     my $targetCue =
$allMessages{$messageID}
{"targetCue"};
     for my $featureLabel(sort(keys(%
{$directFeaturesOld{$messageID}})))
     {
         my $value =
$directFeaturesOld{$messageID}
{$featureLabel};
         if ($value) #don't print 0
values (sparsity)
         {
          print(OUT "$messageID,
$allFeatures{$featureLabel},$value,
$targetCue\n");
         }
     }
    }

    close(OUT);
}

# print_hashes:
# Input: None.
# Output: None.
# Effects: prints to DEBUG file the
end results of the global hashes.
sub print_hashes #expects no args
{
    foreach my $key (sort(keys
%allUsers))
    {
     foreach my $subkey (sort(keys %
{$allUsers{$key}}))
     {
         print(DEBUG "allUsers{$key}
{$subkey} : $allUsers{$key}
{$subkey}\n");
      }
    }

    print(DEBUG "allMessages: \n");
    foreach my $key1 (sort(keys
%allMessages))
    {
     foreach my $subkey1 (sort(keys %
{$allMessages{$key1}}))
     {
        print(DEBUG
"allMessages{$key1}{$subkey1} :
$allMessages{$key1}{$subkey1}\n");
     }
     print(DEBUG "allMessages{$key1}
{guessers} : @{$allMessages{$key1}
{\"guessers\"}}\n");
    }

    print (DEBUG "allWords:\n");
    foreach my $key (sort(keys
%allWords))
    {
     print (DEBUG "allWords{$key}:
$allWords{$key}\n");
    }

    print (DEBUG "allBigrams:\n");
    foreach my $key (sort(keys
%allBigrams))
    {
       print(DEBUG
"allBigrams{$key}:
$allBigrams{$key}\n");
    }

    print (DEBUG "allTrigrams:\n");
    foreach my $key (sort(keys
%allTrigrams))
    {
     print(DEBUG "allTrigrams{$key}:
$allTrigrams{$key}\n");
    }

    print(DEBUG
"directFeaturesOld:\n");

    foreach my $key (sort(keys
%directFeaturesOld))
    {
     foreach my $subkey (sort(keys %
{$directFeaturesOld{$key}}))
     {
        print(DEBUG
"directFeaturesOld{$key}{$subkey}:
$directFeaturesOld{$key}
{$subkey}\n");
     }
    }

    print(DEBUG
"directFeaturesNew:\n");

    foreach my $key (sort(keys
%directFeaturesNew))
    {
     foreach my $subkey (sort(keys %
{$directFeaturesNew{$key}}))
     {
        print(DEBUG
"directFeaturesNew{$key}{$subkey}:
$directFeaturesNew{$key}
{$subkey}\n");
     }
    }

    print(DEBUG "Features currently
extracted LIST:\n");
    foreach my $key
(@allFeaturesList) # ok, not a hash,
but still
    {
     print(DEBUG "allFeaturesList:
$key\n");
    }

    print(DEBUG "Features currently
extracted HASH:\n");
    foreach my
$key(sort(keys(%allFeatures)))
    {
     print(DEBUG "allFeatures{$key}
id is: $allFeatures{$key}\n");
    }

}

# SUB initialize_globals
# Input: None.
# Output: None.
# Effects: Initializes the global
variables, including hashes and
@socialCues
# Remarks: Edit @socialCues if
changing socialCues to parse.
sub initialize_globals #takes no
inputs, to be called at the start of
the program
{
  #initialize fields used by the
entire script (less gross to me than
passing copies and references all
over the place)
    if ($debugging) { print(DEBUG
"initialize_globals\n");}
    @socialCues = ("confidence",
"deception", "disbelief",
"embarrassment", "formality",
"persuading", "politeness",
"rudeness");
    %allUsers = (); #associate user
name with 5 things: totalMessages,
totalCreated, totalGuesses,
guessedCorrectly, createdCorrectly
    %allMessages = (); # maps message
id's with the raw data extracted from
the input files (such as message,
creator, timesGuessedTotal,
timesGuessedCorrectly, times guessed
each of the social cues, targetCue

    # feature hashes: associate
message id's with the features that
can be directly extracted from the
input (does not include part of
speech or mutual information
    %directFeaturesOld = (); # the
features originally extracted
    %directFeaturesNew = (); # the
easiest new features (including
elipses, clause size, subclauses)

    @allFeaturesList =
sort(("exclamMarks", "mainClauses",
"punctMarks", "questionMarks",
"typesToTokensRatio", "wordTokens",
"wordTypes", "QMtoEMRatio",
"elipses", "elipsesRun",
"interrobangs", "mainClausesAv",
"accuracy", "precision")); # lists
all the feature labels currently
being extracted
    %allFeatures = (); # associate
feature label with feature ID
       # including unigrams, bigrams,
and trigrams that appear more than
once in the whole input files

    $totalWordCount = 0; # the number
of words encountered
    $totalUniqueWordCount = 0; # the
number of unique words encountered
(only counts each word once

    # the grams
    %allWords = (); # maps words to
the number of times they appear
    %allTrigrams = ();
    %allBigrams = ();
}

# SUB process_created
# Input: None.
# Output: None.
# Effects: Process the created
messages file, filling in data for
%allMessages
#   and %allUsers.
sub process_created
{
    print("processing created
file...\n");
    if ($debugging) { print(DEBUG
"-----process_created-----\n");}
    # line format: message_id message
time creator difficulty session_id
set_social_tag picture_file
times_guessed times_guessed_correctly
flags

    open(INFILE, "$opt_createdinput")
|| die("Couldn't open createdinput
file $opt_input\n");
    my @infilelines = <INFILE>;
    shift(@infilelines); #remove
first line which is always header
    close(INFILE);

    #if ($debugging) { print(DEBUG
"infilelines: BEGIN @infilelines
END\n");}
    #my $index=0;
    #foreach $fileline (@infilelines)
    print(DEBUG "FILE LINES: \n");

    for (my $index = 0; $index <
scalar(@infilelines); $index++)
    {
     my $fileline =
$infilelines[$index];
     if ($debugging) { print(DEBUG
"line$index: $fileline\n"); }

     my $message_id, $message,
$creator, $difficulty, $session_id,
$target_tag, $picture_file,
$times_guessed,
$times_guessed_correctly;
     my @line_entries;


     # get the info available in the
line
     chomp($fileline);
     @line_entries = split(/\t/,
$fileline);

     if (scalar(@line_entries) <= 3)
#if the line appears to have 2 or
fewer elements, this is probably due
to a \n in the body of a message
     {
         print(DEBUG "Warning! short
line gross times\n");
```

```
        #solution: merge this line
with the next, and skip the next line
by incrementing the index (I know
it's dirty)
        my $next_line =
$infilelines[$index+1];
        chomp($next_line);
        print(DEBUG "next line:
$next_line\n");
        my @next_line_entries =
split(/\t/, $next_line);
        @line_entries =
(@line_entries, @next_line_entries);
        $index ++; #increment index
one extra so as to skip the next line

        #and fix the message entry
        $line_entries[1] .=
$line_entries[2];
        $line_entries[2] =
$line_entries[3];
        $line_entries[3] =
$line_entries[4];
        $line_entries[4] =
$line_entries[5];
        $line_entries[5] =
$line_entries[6];
        $line_entries[6] =
$line_entries[7];
        $line_entries[7] =
$line_entries[8];
        $line_entries[8] =
$line_entries[9];
        $line_entries[9] =
$line_entries[10];
        $line_entries[10] =
$line_entries[11];
        $line_entries[11] =
$line_entries[12];
        #$line_entries[12] =
$line_entries[13];
    } #now I feel all icky

    if ($debugging) {
        my $i = 0; #print("\n");
        foreach (@line_entries) {
         print(DEBUG
"line_entries$i: $_\n");
         $i++;
        }
    }

    $message_id = $line_entries[0];
    $message = $line_entries[1];
    $creator = $line_entries[3];
#skip time_stamp (irrelevant)
    $difficulty = $line_entries[4];
#not planning on using, but maybe
    #skipping session_id
(irrelevant)
    $target_tag = $line_entries[6];
#the social tag set by the message
creator (not necessarily "correct"
depending on the vote system)
    $picture_file =
$line_entries[7]; #if we decide to
separate out pictures
    $times_guessed =
$line_entries[8];
    $times_guessed_correctly =
$line_entries[9];
    # and skipping flags (10 and on)
as irrelevant

    # count user statistics
    &initializeUser($creator);
    $allUsers{$creator}
{"totalMessages"} += 1;
    $allUsers{$creator}
{"totalCreated"} += 1;


&initializeMessageFeatures($message_i
d);
    $allMessages{$message_id}
{"message"} = $message;
    $allMessages{$message_id}
{"targetCue"} = $target_tag;
    $allMessages{$message_id}
{"creator"} = $creator;
    $allMessages{$message_id}
{"difficulty"} = $difficulty;

    #$index++;
   }
}

# SUB process_guesses
# Input: None.
# Output: None.
# Effects: Process the guessed
messages file, filling in data for
%allMessages
#   and %allUsers.
# Remarks: Call after
process_created, but be aware that
messages may
#   (shouldn't, but may) exist in
guesses that did not exist in
created.
sub process_guesses
{
   if ($debugging) { print(DEBUG
"\n\n-----process_guesses-----\n");}
   print("processing guesses
file...\n");

   open(INFILE, "$opt_guessedinput")
```

```
|| die("Couldn't open guessed input
file $opt_input\n");
    my @infilelines = <INFILE>;
    shift(@infilelines); #remove
first line which is always header
    close(INFILE);

    #if ($debugging) { print(DEBUG
"infilelines: @infilelines\n");}
    print(DEBUG "FILE LINES: \n");

    foreach my $fileline
(@infilelines)
    {
     my @lineEntries;
     # get the info available in the
line
     chomp($fileline);
     @lineEntries = split(/\t/,
$fileline);

     if ($debugging) { print(DEBUG
"line: $fileline\n"); }
     # expected line format:
0.guessID, 1.messageID, 2.time,
3.guesser, 4.session,
5.correctSocialTag,
6.guessedSocialTag,
7.guessedCorrectly(0 or 1)
     # guessID, time, session, and
guessedCorrectly are irrelevant

     foreach my $cell (@lineEntries)
     {
         if ($debugging)
{ print(DEBUG "cell: $cell\n");}
     }
     my $messageID, $guesser,
$targetSocialTag, $guessedSocialTag;

     $messageID = $lineEntries[1];
     $guesser = $lineEntries[3];
     $targetSocialTag =
$lineEntries[5];
     $guessedSocialTag =
$lineEntries[6];
     if (!
(&checkTag($guessedSocialTag)) || !
(&checkTag($targetSocialTag)))
     {
         print(DEBUG "target:
$targetSocialTag and guessed:
$guessedSocialTag\n");
         print(DEBUG "skipping to
next entry\n");
         next; # don't include lines
where the social tag is not under
consideration, but don't crash the
script
```

```
         }

         &initializeUser($guesser);
#because there are guessers who
aren't creators, and possibly
creators who aren't guessers
         $allUsers{$guesser}
{"totalGuesses"}+=1;
         $allUsers{$guesser}
{"totalMessages"}+=1;


&initializeMessageFeatures($messageID
); # just in case
         $allMessages{$messageID}
{$guessedSocialTag}+=1;
         $allMessages{$messageID}
{"totalGuesses"}+=1;
         push(@{$allMessages{$messageID}
{"guessers"}}, $guesser);
         if ($targetSocialTag eq
$guessedSocialTag) # guess correctly
         {
              $allMessages{$messageID}
{"totalCorrectGuesses"}+=1;
              $allUsers{$guesser}
{"guessedCorrectly"}+=1;
         }
         else # guessed incorrectly
         {
              #$allMessages{$messageID}
{$guessedSocialTag} += 1;
         }
       }

}

# SUB checkTag
# Input: string $tag
# Output: true (1) if the tag passed
is one of the 8 being checked for
#          false (0) otherwise
# Effects: Prints debug statements to
DEBUG file
sub checkTag #expects tag as a string
{
    my $tag = $_[0];
    foreach my $truetag (@socialCues)
    {
     if ($tag eq $truetag) {
         print(DEBUG "social tag $tag
ok\n");
         return 1; # tag is ok
     }
    }
    print(DEBUG "oops tag $tag is not
an expected social cue\n");
    return 0; # false, tag is invalid
```

```
#if (!($tag eq "deception" || $tag eq
"persuading" || $tag eq "confidence"
|| $tag eq "formality" || $tag eq
"politeness" || $tag eq "rudeness" ||
$tag eq "embarrassment" || $tag eq
"disbelief")) #tag is not any of the
8, return false
#     {     print(DEBUG "oops tag $tag
is not an expected social cue\n");
#     return 0;}
#     else
#     {     if ($debugging)
{ print(DEBUG "social tag $tag
ok\n");}
#     return 1;}
}


# SUB initializeMessageFeatures
# Input: int $message_id
# Output: None.
# Effects: Intializes some of the
data required to calculate features,
without
# overwriting it if it already
exists.
# Remarks:
#    Current features extracted:
#    (2) word types in message (unique
words in message)
#    (3) word tokens in message (total
words in message)
#    (4) type to token ratio (use type
and token counts to calculate)
#    (5) # of punctuation marks in
message (can include ellipsis)
#    (5a) # of questions marks in
message
#    (5b) # of exclamation marks in
message
#    (6) # of separate
sentences/questions in message
#    (7) average word length per
message
#    (8) mean log frequency of words
used (compared against words used
#         in all messages)
#    (9) through (n) count of
vocabulary item used (doesn't include
words
#         only used once)
#    (b1) through (bn) count of
bigrams used (doesn't include bigrams
only
#         used once)
#    (t1) through (tn) count of
trigrams used (doesn't include
trigrams
#         only used once)
```

```
sub initializeMessageFeatures
#expects message_id
{
    my $messageID = $_[0]; # fetch
input argument
    if ($debugging) { print(DEBUG
"*initializeMessageFeatures:
messageID: $messageID\n");}

#     if (!exists
$allMessages{$messageID})    {
#     %{$allMessages{$messageID}} =
();
#     }

    if (!exists
$allMessages{$messageID}
{"totalGuesses"})     {
      $allMessages{$messageID}
{"totalGuesses"} = 0;
    }
    if (!exists
$allMessages{$messageID}
{"totalCorrectGuesses"})     {
      $allMessages{$messageID}
{"totalCorrectGuesses"} = 0;
    }
    #initialize the times guessed
each of the 8 social categories
    if (!exists
$allMessages{$messageID}
{"formality"})     {
      $allMessages{$messageID}
{"formality"} = 0;
    }
    if (!exists
$allMessages{$messageID}
{"politeness"})     {
      $allMessages{$messageID}
{"politeness"} = 0;
    }
    if (!exists
$allMessages{$messageID}
{"deception"})     {
      $allMessages{$messageID}
{"deception"} = 0;
    }
    if (!exists
$allMessages{$messageID}
{"confidence"})     {
      $allMessages{$messageID}
{"confidence"} = 0;
    }
    if (!exists
$allMessages{$messageID}{"rudeness"})
{
      $allMessages{$messageID}
{"rudeness"} = 0;
    }
    if (!exists
$allMessages{$messageID}
{"persuading"})     {
      $allMessages{$messageID}
{"persuading"} = 0;
    }
    if (!exists
$allMessages{$messageID}
{"disbelief"})     {
      $allMessages{$messageID}
{"disbelief"} = 0;
    }
    if (!exists
$allMessages{$messageID}
{"embarrassment"})     {
      $allMessages{$messageID}
{"embarrassment"} = 0;
    }
    if (!exists
$allMessages{$messageID}{"guessers"})
{
      @{$allMessages{$messageID}
{"guessers"}} = ();
    }
}

# Initializes the 5 relations for a
given user (if they haven't already
been).
# It is possible for this subroutine
to be called multiple times on a
given
# user.
# Thus, calling this subroutine
before modifying the data associated
with a
# user name is safe even if a user
has already been initialized, and
saves the
# hastle of multiple existance
checks.
sub initializeUser #expects the
username
{

    my $username = $_[0]; # fetch
input argument

    if ($debugging) { print(DEBUG
"*initializeUser: username:
$username\n");}

    if (!exists $allUsers{$username})
    {
     %{$allUsers{$username}} = ();
    }
    if (!exists $allUsers{$username}
{"totalMessages"}) #1
    {
```

```
      $allUsers{$username}
{"totalMessages"} = 0;
    }
    else
    {
#     if ($debugging){ print(DEBUG
"oops $username totalMessages already
initialized");}
    }

    if (!exists $allUsers{$username}
{"totalCreated"}) #2
    {
      $allUsers{$username}
{"totalCreated"} = 0;
    }
    else
    {
#          if ($debugging)
{ print(DEBUG "oops $username
totalCreated already initialized");}
    }
    if (!exists $allUsers{$username}
{"guessedCorrectly"}) #4
    {
      $allUsers{$username}
{"guessedCorrectly"} = 0;
    }
    else
    {
#     if ($debugging){ print(DEBUG
"oops $username guessedCorrectly
already initialized");}
    }
    if (!exists $allUsers{$username}
{"createdCorrectly"}) #5
    {
      $allUsers{$username}
{"createdCorrectly"} = 0;
    }
    else
    {
#     if ($debugging) { print(DEBUG
"oops $username createdCorrectly
already initialized"); }
    }
}

# SUB update_allFeatures
# Input: None.
# Output: None.
# Effects: Updates global
@allFeaturesList with the
uni/bi/trigrams
#   that appear more than once in the
whole input.
#   Updates global %allFeatures hash
with the feature ID associated with
each feature label found in
```

```
allFeatursList
# Remarks: Best called after
%allWords, %allBigrams, %allTrigrams
are updated
#   for every message.  Could check
for repeats, but it would be slower.
sub update_allFeatures
{
    for my $word (keys(%allWords))
    {
     if ($allWords{$word} > 1)
     {
         print(DEBUG "word:
$word\n");
         push(@allFeaturesList,
"word:".$word);
     }
    }

    for my $bigram
(keys(%allBigrams))
    {
     if ($allBigrams{$bigram} > 1)
     {
         push(@allFeaturesList,
"bigram:".$bigram);
     }
    }

    for my $trigram
(keys(%allTrigrams))
    {
     if ($allTrigrams{$trigram} > 1)
     {
         push(@allFeaturesList,
"trigram:".$trigram);
     }
    }
    #finally, sort at the end
    @allFeaturesList =
sort(@allFeaturesList);

    my $featureID = 1;
    for my $featureLabel
(@allFeaturesList)
    {
     $allFeatures{$featureLabel} =
$featureID;
     $featureID++;
    }
}

# Grams: Unigrams, Bigrams and
Trigrams:
#   where
#      unigram: a single words
#      bigram: sequence of 2 words
#      trigram: sequence of 3 words
#   Currently, disregarding
punctuation and capitalization
#   For example, "it's" and "its" are
the same (maybe fix).
#   "I" and "i" are the same (maybe
fix).


# SUB update_allTrigrams
# Input: int $messageID
# Output: None.
# Effect: update global %allTrigrams
hash to include the trigrams
extracted from
#   message associated with
$messageID.
# Remarks: Should only be called once
per message
sub update_allTrigrams{
    my $messageID = @_[0];
    print(DEBUG "update trigrams for
message id: $messageID\n");
    my @messageWords =
remove_nonwords( get_word_list($allMe
ssages{$messageID}{"message"}));
    my $index, $trigram;
    my %trigrams =
get_trigram_list(@messageWords);

    foreach $trigram
(sort(keys(%trigrams))) {
      print(DEBUG "   trigram is
$trigram\n");

if(exists($allTrigrams{$Trigram})){
        $allTrigrams{$trigram} +=
$trigrams{$trigram};
      }else{
        $allTrigrams{$trigram} =
$trigrams{$trigram};
      }
    }
    print(DEBUG "done updating
trigrams for id: $messageID\n");
}

# SUB get_trigram_list
# Input: A list of words
(@messageWords).
# Output: Hash %trigrams associating
each trigram present in the message
#   of $messageID with the number of
times it appears in the message
# Effects: None (besides print to
DEBUG).
sub get_trigram_list{
    #my $messageID = @_[0];
    print(DEBUG
"get_trigram_list\n");
```

```
    my @messageWords = @_; # fetch
input list
        #&get_word_list($allMessages{$
messageID}{"message"});
    my %trigrams = ();
    my $index;

    # separate into groups of 3
words, separated by a +
    # BEGIN = beginning of message
    # END = end of message
    # currently, punctuation is
removed (all words simply treated as
one long string)
    for($index = 1; $index <
$#messageWords; $index++){
      #print(DEBUG
"messageWords[$index] is
$messageWords[$index]\n");
      # if second word, trigram is
BEGIN+$word0+$word1
      if($index == 1){
          $trigram = "BEGIN\+".
$messageWords[$index-1]."\+".
$messageWords[$index];
      }else{
          $trigram =
$messageWords[$index-2]."\+".
$messageWords[$index-1]."\+".
$messageWords[$index];
      }
      #@trigrams = (@trigrams,
$trigram);
      !(exists $trigrams{$trigram})?
          $trigrams{$trigram}=1 :
          $trigrams{$trigram}++ ;
    }

    # do last word ($wordindex-
1+wordindex+END)
    $trigram =
$messageWords[$#messageWords-1]."\+".
$messageWords[$#messageWords]."\
+END";
    print(DEBUG "trigram is
$trigram\n");
    #@trigrams = (@trigrams,
$trigram);
    !(exists $trigrams{$trigram})?
     $trigrams{$trigram} = 1 :
     $trigrams{$trigram}++   ;

    return %trigrams;
}

# SUB update_allBigrams
# Input: messageID
# Output: none.
# Effect: Update global variable
```

```
%allBigrams with the bigrams
extracted
#   from the message of messageID.
# Remarks: Should only be called once
per message.
sub update_allBigrams{
  my $messageID = @_[0];
  print(DEBUG "messageID:
$messageID\n");
  my @messageWords =
get_word_list($allMessages{$messageID
}{"message"});
     # split on pattern (one or more
of any white space)
  #my $index, $bigram;
  my @bigrams = ();
  print(DEBUG
"allMessages{$messageID}
{\"message\"}:
$allMessages{$messageID}
{\"message\"}\n");

  for $word (@messageWords)  {
      print(DEBUG " word: $word\n");
}

  @messageWords =
&remove_nonwords(@messageWords);
#should be redundant
  %bigrams =
&get_bigram_list(@messageWords);

  foreach my $bigram
(sort(keys(%bigrams))){
      print( DEBUG "bigram is
$bigram\n");
      if(exists($allBigrams{$bigram})
){
        $allBigrams{$bigram} +=
$bigrams{$bigram};
      }else{
        $allBigrams{$bigram} =
$bigrams{$bigram};
      }
  }
}

# Input: a list of words
(@messageWords).
# Output: Hash of bigrams to the
number of times apppeared in the
input.
# Effects: None.
sub get_bigram_list{
  my @messageWords = @_;
  my %bigrams = ();
  my $index;

  # separate into groups of 2 words,
separate by a +
  # BEGIN = beginning of message
  # END = end of message
  # currently, no punctuation is used
(all words simply treated as one long
string)

  for($index = 0; $index <=
$#messageWords; $index++){
    #print(DEBUG
"messageWords[$index] is
$messageWords[$index]\n");
    # if first word, bigram is BEGIN+
$word0
    if($index == 0){
      $bigram = "BEGIN\+".
$messageWords[$index];
    }else{
      $bigram = $messageWords[$index-
1]."\+".$messageWords[$index];
    }
    !(exists $bigrams{$bigram})?
     $bigrams{$bigram}=1 :
     $bigrams{$bigram}++ ;
  }

  # do last word ($wordindex+END)
  $bigram =
$messageWords[$#messageWords]."\
+END";
    !(exists $bigrams{$bigram})?
      $bigrams{$bigram}=1 :
     $bigrams{$bigram}++;

  return %bigrams;
}

# Input: string $message
# Output: A list of words as
separated by one or more
#   white spaces
sub get_word_list{
    my $message = @_[0];
    print(DEBUG "before:
$message\n");
    $message =~ s/[^\w\s]+//g;#
remove ALL punctuation (not
alphanumeric or not whitespace), not
just first occurence (g)
    print(DEBUG "after: $message\n");
    return split('\s+', $message);
}


# takes a list of words as input,
removes non word things, returns the
new list
# pass by copy and does not modify
original input
```

```
sub remove_nonwords{
  my @messageWords = @_;

  my @messageWordsFiltered = ();

  # get rid of
  for($index = 0; $index <=
$#messageWords; $index++){
    if($messageWords[$index] =~ /\w/)
{
      @messageWordsFiltered =
(@messageWordsFiltered,
$messageWords[$index]);
    }
  }

  return @messageWordsFiltered;
}

### Calculation functions:
# (Often short) functions that
calculate various features from a
# given message.  Cleans up the code
considerably to put them down
# here.  Eases testing.

# SUB get_num_qm
# Input: string $message (as is, no
preprocessing required)
# Output: int number of question
marks contained in $message
# Effects: None.
# Remarks: The previous methodology
was bugged.
sub get_num_qm
{
    my $message = @_[0]; # fetch
input
    my $num = 0;
    while($message =~ /\?/g){$num++;}
    return $num;
}

# SUB get_num_em
# Input: string $message (as is, no
preprocessing required)
# Output: int number of exclamation
marks contained in $message
# Effects: None.
# Remarks: The previous methodology
was bugged.
sub get_num_em
{
    my $message = @_[0]; # fetch
input
    my $num = 0;
    while($message =~ /\!/g){$num++;}
    return $num;
}
```

```
# SUB get_num_ib
# Input: string $message (as is, no
preprocessing required)
# Output: int number of interrobangs
contained in $message
# Effects: None.
# Remarks: The previous methodology
was bugged. Interrobangs are
#   considered to be the substring
'?!' and '!?' (order is irrelevant).
#   Overlaps are counted.  For
example, '?!?' would count as 2
interrobangs
#   and '?!?!' would be 3.
sub get_num_ib
{
    my $message = @_[0]; # fetch
input
    my $num = 0;
    #while($message =~ /\!\?|\?\!/)
{$num++;}
    $num =()= $message =~ /(\?\!)/g;
# isn't that beautiful perly code!
    $num +=()= $message =~ /(\!\?)/g;
    # ok, what that does is (starting
from the right), match $message with
'?!' and assign the result to an
empty list, and assign that to a
scalar context, so it ends up
counting the number of times '?!'
substring appears in $message,
including overlap!  Then I add the
result of matching '!?' for
completeness sake.
    return $num
}

# SUB get_qm_to_em
# Input: string $message (as is, no
preprocessing required)
# Output: rational number expressing
the ratio of question
#   marks to exclamation marks.
(qm/em)
# Effects: None.
# Remarks: Uses &get_num_qm and
&get_num_em.  If num_em is 0,
#   returns 0.
sub get_qm_to_em
{
    my $message = @_[0]; # fetch
input
    my $em = &get_num_em($message);
    if ($em == 0){ # woops divide by
0
      return 0;
    }
    else
    {
      return (&get_num_qm($message)/
$em);
    }
}

# SUB get_num_elipses
# Input: string $message (as is, no
preprocessing required)
# Output: int number of elipses in
the message
# Effects: None.
# Remarks: An elipses is considered
to be 2 or more consecutive
#   periods (ie. '.').  Overlap is
not counted, so
#   '..' is 1, '...' is also 1, and
'.. ...' is 2.
sub get_num_elipses
{
    my $message = @_[0]; # fetch
input
    my $num = 0;
    while($message =~ /\.\.+/g){$num+
+;} #match one and at least one '.'
    return $num;
}

# SUB get_longest_elipses_run
# Input: string $message (as is, no
preprocessing required)
# Output: int length of the longest
run of elipses in the message
# Effects: None.
# Remarks: An elipses is considered
to be 2 or more consecutive
#   periods (ie. '.').
sub get_longest_elipses_run
{
    my $message = @_[0]; # fetch
input
    my @elipses = $message =~ /\.\.
+/g;
    @elipses = sort(@elipses); #
since the elements are just .. of
various length, has the nice side
effect of doing exactly what I want:
sort by length
    # Only, the longest one is at the
end of the list.
    my $longest =
$elipses[$#elipses];
    my $num = 0;
    while($longest =~ /\./g){$num++;}
#and count the dots
    return $num;
}
```