Evaluating Machine Learning Algorithms and the ClueWordSummarizer for Email Thread Summarization

Jan Ulrich
Department of Computer Science
University of British Columbia
2366 Main Mall
Vancouver, BC V6P 5H9
ulrichj@cs.ubc.ca

Abstract

Machine learning summarizers and the ClueWordSummarizer are two different approaches to summarizing email threads. These approaches are compared to show which is a more viable solution under different conditions. The current results show that both approaches perform comparably. With some possible improvement to features, machine learning has the potential to create more accurate summaries. Future directions for email thread summarization are discussed.

1 Introduction

Email has become a part of most people's everyday lives. With its widespread use, using it as effectively as possible has become important. Summarization provides one aspect in reducing the information overload. A summary of an email thread limits itself to the salient parts of the email conversation. Email threads are of particular interest to summarization because they provide lots of structural redundancy due to their conversational nature.

Although summaries are useful, they still cannot replace the original emails in all cases. This is because the algorithms don't always perform as human summarizers would. Also, usually the original message contains additional information which is lost in the summary. Therefore the summaries are used as a tool in certain situations, not to replace reading actual emails. Some of these situations include getting an overview of what has happened when your mailbox is overwhelmingly full. Summaries also are a type of memory. A written summary can help you remember what you had been discussing previously. This is especially important in the business world, where summaries form a corporate memory. The content of previous business decisions can be archived in email thread summaries. Summaries are also useful in investigations. They provide an easy way of indexing, by allowing the reader to get an overview of the conversation and then providing a link to the original emails. They can also serve as a tool to highlight the important parts of a conversation. When a new email arrives in the mailbox, a summary of the conversation can be generated and any summary sentence highlighted in the incoming email. It can also serve as a kind of writing checker, by summarizing an outgoing message that is part of a thread, and then showing you which sentences would not be included in a summary, allows you to remove some to make the message more concise.

Summarization can be divided into two different types: extractive and abstractive. In extractive summarization the important sentences are copied from the original document to form the summary. This means that nothing is changed and a summary is a strict subset of the original document. In an abstractive summary, the document is rewritten in a more concise form. In an extractive summary since the sentences are just picked from the original document, they may not flow and the coherence will be very low. The sentences can be rearranged so the document make more sense. Although an abstractive summary seems like a better solution, the problem is much more complex and therefore extrac-

tive summarization has been the standard in multi-document summarization.

2 Work on Email Thread Summarization

There has been much work done on email thread summarization. Most of the work uses techniques learned in multi-document summarization and applies them to email summarization by including email specific elements. The Clue Word Summarizer (CWS)(Carenini et al., 2007) takes advantage of the email thread structure when creating the emails. Other attempts have used a machine learning approach to summarization. Here a classifier is trained to do the sentence extraction using a variety of features (Rambow et al., 2004). These are two very different approaches to the same email summarization problem. The machine learning approach requires a corpus for the training. Such a corpus has become available and thus machine learning is now a feasible solution to the email thread summarization problem. In this paper I will perform an in depth comparison between these two techniques. For future development it is worth knowing which one performs better. There is even the possibility of combining the two techniques since clue score could be a feature in the machine learning technique.

CWS was developed by Xiaodong Zhou at the University of British Columbia (Carenini et al., 2007). For this research the full system was available for evaluation. The machine learning summarization system built by (Rambow et al., 2004) serves as a good model for machine learning summarization. The first step was to build summarizer modeled after this one and then compare the performances.

3 ClueWordSummarizer

CWS requires no previous training data or knowledge about the email topic. It uses the conversation structure to provide an accurate summary. Therefore the first step is to create the email conversation structure. To do this, a quotation graph is built. This is done by taking advantage of the fact that most email clients will paste the previous email into the current one as a quote. Therefore by looking at the quote structure the email hierarchy can be created.

3.1 Hidden Emails

The inclusion of quoted previous emails allows for the recovery of hidden emails. These are emails that the user has deleted from their mailbox, but that can still be recovered because they are quoted in a following email. These emails are extracted and added to the hierarchy as if they were still in the mailbox.

3.2 Fragment Quotation Graph

The fragment quotation graph is a refined version of the message quotation graph. In this graph the nodes are more detailed since they are fragments which are parts of an email message. A fragment is derived from the fact that users sometimes reply to specific parts of the previous email. They do this by writing their reply directly beneath different parts of the quoted email. The algorithm then divides the original message into different fragments depending on the position of the reply of the user. Analysis of the email thread at the fragment level allows for a more accurate conversation structure.

3.3 Clue Word

The basis for CWS are clue words. Clue words are words that are important to the conversation. People tend to use the same key words from the original email in their replies. Therefore a clue word is a word that is repeated in the child and parent node in the fragment quotation graph (excluding stopwords). These clue words should be included in a summary.

3.4 The Overall System

CWS first computes the fragment quotation graph including the hidden emails. From this graph the clue words can be identified. Then each sentence is given a clue score based on the number of clue words it contains. This score is then used to select which sentences to extract for the summary.

4 A Machine Learning Summarizer

Another approach has been to use machine learning to summarize email threads. (Rambow et al., 2004) have used email specific features to classify sentences for extraction based summarization. They combined successfully used features from text summarization with email specific features. Then using a corpus, a collection of training data, they trained a

rule based binary classifier to determine which sentences would be included in the summary. Their work showed that using email specific features was indeed useful for summarization.

5 Overview

In this paper I will compare CWS to a machine learning summarizer. For future research it will be useful to know which one performs better and which has the most potential. The machine learning approach requires a corpus for training, so this will also tell if spending time collecting a corpus is a justified endeavor. The rest of the paper is laid out as follows: the machine learning email thread summarization system is described in section 6. In section 7 the corpus and annotations that were used are described in detail. Section 8 explains all the different features that were used and section 9 describes the different machine learning classifiers. Then section 10 describes how all these pieces fit together. Section 11 outlines the evaluation that was performed and section 12 & 13 provide the results. Section 14 talks about the implications of these results and section 15 goes over some of the challenges in working on this project. Finally, section 16 introduces future work on this project. Section 17 contains a pointer to the Enron corpus and some of the source code for the project

6 Machine Learning Email Thread Summarization

A machine learning summarizer uses a classifier that creates a sentence importance score for each sentence. This score is then used to decide which sentences to include in the summary. Ten features were generated from the email corpus that was used. The summarizer used two software packages in its implementation. MEAD (Radev et al., 2004) was used as the summarization framework. WEKA (Witten et al., 1999) was used as the implementation of machine learning classification algorithms. These two packages were combined to create several different machine learning summarizers that were compared to CWS.

6.1 MEAD

MEAD is a open source multi-document summarization framework. It works in three components. First features are generated, then a classifier scores sentences, and finally a reranker produces the final summary. The system is build so that each one of these components can be changed independently for development and analysis.

6.2 WEKA

WEKA is a collection of machine learning algorithms developed by the University of Waikato. It is implemented in Java and can be used to train and test a variety of classification algorithms. It is provided as open source software and is well-suited for comparing different machine-learning techniques.

7 The Corpus

In order to do machine learning summarization, a corpus of email threads is needed for training the classifiers. Email data can be hard to obtain for research purposes because of privacy concerns. People are not inclined to donate their private emails, even for research. However the email research community has been blessed with an unaltered, real email corpus. This is due to the Federal Energy Regulatory Commission releasing the Enron corporation's emails to the public during the investigation into accounting fraud. This corpus contains a huge number of messages sent or received by Enron employees. From these emails, 39 email threads were selected from the 10 largest email inbox folders such that each thread contained at least four emails.

These email conversations were annotated using manual sentence extraction. The 50 annotators that were recruited for the study were undergraduates and graduate students from the University of British Columbia. Their majors covered various disciplines including Arts, Law, Science, and Engineering. The variety of backgrounds for the summarizers provides for less biased summaries.

Each annotator had to summarize four email threads by selecting which sentences to include. Each thread was therefore annotated by five annotators. The annotators were asked to pick 30% of the original sentences such that the summary contained the overall information in the email and could

be understood without referencing the original email thread. Sentences were labeled as either *essential* or *optional*, where an essential sentence is vital to the understanding of the summary and an optional sentence elaborates on the meaning of the conversation but is only included if there is space in the summary.

7.1 GSValue

A GSValue score was computed for each sentence in the corpus. This score represents the annotators' combined feeling about the importance of the sentence. Since each sentence was either selected as *essential* or *optional* or not selected at all by each annotator, a score is calculated taking into account each annotator's remarks. The score weighs a vote for *essential* as three points and a vote for *optional* as one point. Since there are five annotators, the GSValue score ranges from zero to fifteen.

$$GSValue = 3 * \#ESS + \#OPT \qquad (1)$$

8 Features

The features are the information from each email thread that is actually used to select which sentences are included in the summary. Choosing features is like choosing what aspects of sentences are important in summaries. The features that have been included are those that have been shown to work in multi-document summarization as well as some that are email specific. In the work by (Rambow et al., 2004) including email specific features significantly improved performance. 10 features were used in the study and they are listed below. These features are calculated for each sentence and are therefore in relation to that sentence.

- *Thread Line Number* - The position of the sentence in the thread.

- *Relative Position in Thread* - The position in the thread as a percentage (Thread Line Number /Number of sentences in the thread).

- *Centroid Similarity* - The cosine similarity of the sentence tf*idf vector to the thread average tf*idf vector. The idf component is computed for the whole corpus.

- *Local Centroid Similarity* - The same as the centroid similarity except that the inverse document frequencies are computed for the current thread instead of the whole corpus.

- *Length* - The number of words in the sentence.

- *TF*IDF Sum* - The sum of the sentence's tf*idf values.

- *TF*IDF Average* - The average of the sentence's tf*idf values.

- *Is Question* - A boolean stating whether the sentence is a question (based on punctuation).

- *Fragment Number* - The temporal position of the current fragment.

- *Relative Position in Fragment* - The temporal position of the current fragment as a percentage (Fragment Number / Total number of fragments).

9 Classifiers

The classifier is the part of the MEAD system which scores each of the different sentences. The features of each sentence are fed into the classifier to create an importance score. The reranker can then adjust the score to fit its own needs, which in this system is to minimize redundancy. The class for classification is GSValue. Since this value can range from zero to fifteen, depending on sentence importance, regression classifiers are chosen. The three classifiers that are compared in this paper are SMOreg, Simple Linear Regression, and Bagging. They are implemented in the WEKA software package.

SMOreg is an optimization algorithm for training support vector regression using a polynomial kernel. The algorithm implements sequential minimal optimization using extreme chunking by converting nominal attributes into binary ones and optimizing the target function for them.

Simple Linear Regression builds a regression model by repeatedly selecting the attribute providing the lowest squared error.

Bagging was done on decision tree classifiers. Bagging is a bootstrapping algorithm which averages the output of several decision trees trained on

random subsets of the training data. It improves stability and accuracy. The decision tree algorithm that was used was REPTree which builds a regression tree using information variance reduction.

10 The Combined Summarization System

The combined summarization system combines the MEAD framework with the WEKA classifiers using the 10 mentioned features. First the email conversations are preprocessed to determine the conversation structure. Hidden emails are identified and extracted from the quotation. The messages are then divided into fragments as described in CWS. When dividing messages into fragments only the text is kept. Any headers or attachments are removed. These fragments are then inserted into the MEAD framework so that each email thread is one cluster. A cluster summary will then be a thread summary. The features listed above are calculated for each sentence. Each sentence is also assigned a class which is the GSValue derived from the annotators' rating. Now these features with class values can be exported to WEKA for training a classifier. An .arff file is created and used to train the corresponding classifier. This classifier is then used in the MEAD summarization system. A new test thread is loaded into MEAD and the features are computed. Now the classifier is run on the list of feature vectors and the sentences are given a score. This score is used by the reranker to generate a final summary with reduced redundancy. This is how an email thread summary is generated using a machine learning summarizer which combines MEAD and WEKA.

11 Evaluation

For the evaluation the generated summaries from the different summarizers were compared to the human generated gold standard. The machine learning summarizers were tested using a 10-fold cross-validation. Thus the 39 email threads were divided into ten tests where 90% of the data was used for training and 10% of the data was used for testing. This meant that there were 10 randomly generated test sets of four threads. This was done for all three different classification algorithms. The 40 different resulting summaries were then compared to the corresponding gold standard. The gold standard summaries were created from the GSValues by using a threshold level of 8. This means that any sentence with a GSValue greater than 8 was included in the summary. The summaries were compared using ROUGE-2 (Lin and Hovy, 2003). This is a bigram based comparison which has been shown to correspond well with human evaluations and is being used in the Document Understanding Conference. The results can be seen in Figure 1. Comparing the actual summaries, instead of the sentence indices, is a good test since this can account for similarities between the sentences.

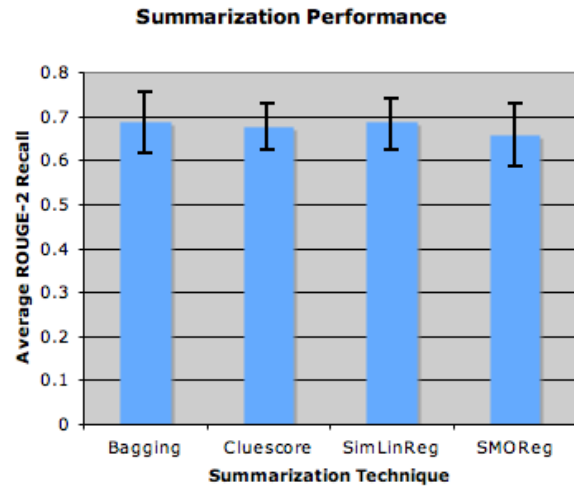

Figure 1: ROUGE-2 comparison between the different machine learning algorithms and the ClueWordSummarizer.

The classifiers can also be evaluated at a lower level. Since we are performing cross-validation, even the GSValues of the test set sentences are available. Thus we can see how accurate the classifiers were in their results. Figure 2 shows the mean absolute error of the algorithm's classification. Since GSValue is a continuous value, this error shows us how far off the classifier was on average.

12 Results

The machine learning summarizer using bagging for classification received the highest ROUGE-2 recall score. However there was no significant difference among any of the tested methods. The ROUGE-2 recall scores are averaged over all 10 cross-validations. The error bars represent the range

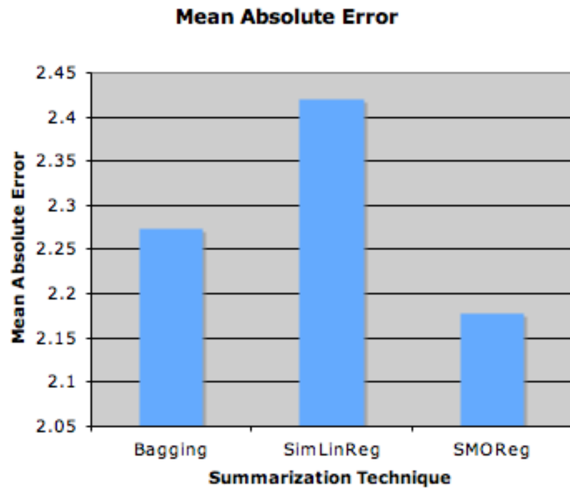

Figure 2: The error in the sentence importance prediction of the different classifiers.

of values that fall into the 95% confidence interval.

These results have some integrity problems. After computing the results it was noticed that the fragments were not strictly in chronological order in the MEAD clusters. This means that features that rely on the ordering of the fragments can have the wrong values. These faulty features should not have a detrimental effect on the results since the algorithms use the features that are most useful. However the results are surely not as good as they could be for the machine learning summarizers. The faulty features had no effect on CWS since no features were used for this approach. This should be promising for the machine learning summarizers since by correcting these features the summarizers' performance would improve.

In figure 2 it can be seen that the different classifiers performed differently for the average classification accuracy. However their ROUGE-2 summary results are very similar. This means that the summary score is not very sensitive to small amounts of error in the classification. In fact the classifier with the smallest amount of error, SMOreg, performed worst in the ROUGE-2 recall (although not significantly). Different evaluations are needed validate the ROUGE-2 scores.

13 Summarizer Results

Summarizer	ROUGE-2 Recall	95% Conf.
Bagging	0.687	0.617 - 0.755
Cluescore	0.677	0.624 - 0.727
SimLinReg	0.686	0.623 - 0.742
SMOReg	0.659	0.586 - 0.730

14 Discussion

The results bode well for CWS, since all systems perform similarly well, but CWS does not need to be trained and therefore does not need a corpus.

Neither of these approaches have been tested for robustness to different kinds of emails. They were designed to summarize email threads, so of course they would not work for single emails. However they were trained and tested on the Enron corpus, and it is not known how well they perform on other corpora. There is surely a difference between a strictly personal email inbox and a business email inbox. The machine learning summarizers could be trained on a different corpus if it was available. However the features might not work as well for such a corpus as different aspects of an email conversation could be important in strictly personal emails. CWS would not have to be changed at all, but in case it doesn't work as well, there is not much that can be done.

This project was successful in creating a first impression of the comparison between CWS and the machine learning summarizers. However, the fact that some of the features were faulty is a major problem for the validity of results. Optimistically one can say that the machine learning algorithms can only get better by fixing the problem, but this is not necessarily true, and one can only be sure when the results are available.

This project did however build a framework for machine learning summarization which can easily be modified and extended. The hard work was accomplished which combined MEAD and WEKA so that additional features and classifiers only need to be plugged into the system. It is also possible to change the evaluation of the summaries. Therefore the real contribution of this project was to build a framework which fosters research.

15 Challenges

There have been a number of challenges in writing this paper that have turned into good learning experiences. Having to create each one of the features by hand proved to be more of a challenge than first expected. Having read many papers on summarization and information retrieval all the traditional features were very familiar. Therefore conceptually the problem of generating the features seemed very simple. For example when actually having generate tf*idf vectors, you realize just how many numbers you are dealing with. I gained perspective by actually generating all these features that I have read so many papers about.

I also saw that although the overall concepts are agreed upon, the details can be very different in different applications. For example term frequency, is a very basic concept in information retrieval. It is just the number of times a term appears in a given document which is normalized by length because documents vary in size. Since I am dealing with email sentences, the tf vector was just the number of occurrences of each word divided by the number of words. However in the MEAD implementation, when a tf vector is used for creating a centroid, it is calculated to be the average number of occurrences of each term in a sentence. That means all the sentences in the document have the same tf vector. The documentation states this works for building a centroid, but it did not work for my features, so sometimes implementations are optimized and are no longer general.

I also had to learn PERL which is a powerful, high level programming language in which MEAD is developed. Whenever you learn a new programming language, it is a lot of work at first and only slowly are you able to write programs. However by the end of this project I am a lot more comfortable in writing PERL and see how useful it can be to do tasks without having to write too much code.

I also had to combine two software packages that were written in different languages: PERL and Java. Combing different interfaces is always challenging, but the result is very powerful since both software packages perform their functions very well. By using software that has been widely used and is understood by many other researchers, my work becomes much more valid. I also learned to use other people's code by trying to understand their documentation, and looking through their source code.

In this project I was able to take the high level plan and implement it by combining two different software packages, as well as learn a new programming language.

16 Future Work

The first thing that has to be fixed is the ordering problem of the fragments for the machine-learning summarizers. This will correct any of the features that depend on the chronology of the fragments, and will therefore make higher quality summaries.

Additional features should also be added to the system. At present there are only two email related features in the system. Three more were used by (Rambow et al., 2004) which can be included in the system. These were the number of direct responses to the current email and the number of recipients. Also the email subject can be compared to the thread's original subject. This last feature accounts for emails that should not have been in the thread because the point of the conversation has been changed and the author also changed the subject line.

As an additional baseline, the standard MEAD multi-document summarizer can also be included. This will compare email thread specific summarizers to more general multi-document summarizers. This would validate the need for additional features that have been found useful in email thread summarization (Rambow et al., 2004).

A variety of evaluation should also be used. The ROUGE-2 evaluation method depends on one gold standard summary to which candidate summaries are compared. However it has been found that human summaries of the same email thread are often very different (Rath et al., 1961) and therefore there should not be one gold standard for each summary. Therefore other evaluation methods should be used as well. Since the summarizers we are comparing are all extraction based summarizers, the sentence index numbers can be used to evaluate the summarizers. The sentences each received a GSValue from the annotators, which represents the importance of the sentence. By adding up all the GSValue points in a summary, a summary can receive a GSscore which

represents it's information content level. This approach is similar to (Nenkova et al., 2007) in that multiple annotator's summaries are used to score the candidate summary. This approach would also eliminate the arbitrary threshold value of 8 that is used to create the gold standard summary for ROUGE-2 evaluations.

It would also be interesting to do a comparison of a wider variety of classification algorithms. WEKA contains implementations of a vast number of classifiers which makes it possible to do such a broad comparison. However most classifiers do not work with ordinal classes which limits the number that can be used. Even so, the ordinal problem can be converted into a binary problem by choosing different thresholds for GSValues. A sentence would then be labeled as included or not included in the summary, and a binary classifier can be used to solve the problem. Local ordinal classification techniques have also been developed which convert an ordinal class into a set of binary classes while preserving the ordering of the original classes (Kotsiantis, 2006). This is also a promising approach to using a wider variety of classifiers.

References

Giuseppe Carenini, Raymond T. Ng, and Xiaodong Zhou. 2007. Summarizing email conversations with clue words. *16th International World Wide Web Conference (ACM WWW'07)*.

Sotiris B. Kotsiantis, 2006. *Local Ordinal Classification*, volume 204/2006 of *IFIP International Federation for Information Processing*, pages 1–8. Springer Boston.

Chin-Yew Lin and E.H. Hovy. 2003. Automatic evaluation of summaries using n-gram co-occurrence statistics. *Proceedings of Language Technology Conference (HLT-NAACL)*.

Ani Nenkova, Rebecca Passonneau, and Kathleen McKeown. 2007. The pyramid method: Incorporating human content selection variation in summarization evaluation. *ACM Trans. on Speech and Language Processing (TSLP)*.

Dragomir Radev, Timothy Allison, Sasha Blair-Goldensohn, John Blitzer, Arda Çelebi, Stanko Dimitrov, Elliott Drabek, Ali Hakim, Wai Lam, Danyu Liu, Jahna Otterbacher, Hong Qi, Horacio Saggion, Simone Teufel, Michael Topper, Adam Winkel, and Zhang Zhu. 2004. MEAD - a platform for multidocument multilingual text summarization. In *Proceedings of LREC 2004*, Lisbon, Portugal, May.

Owen Rambow, Lokesh Shrestha, John Chen, and Chirsty Lauridsen. 2004. Summarizing email threads. *Proceedings of HLT-NAACL 2004*.

G. J. Rath, A. Resnick, and R. Savage. 1961. The formation of abstracts by the selection of sentences. *American Documentation*.

I.H. Witten, E. Frank, L. Trigg, M. Hall, G. Holmes, and S.J. Cunningham. 1999. Weka: Practical machine learning tools and techniques with java implementations. *ICONIP/ANZIIS/ANNES*.

17 Appendix

17.1 The Enron Corpus

The Enron corpus contains a large number of emails from the Enron corporation's former employees that were made public by the accounting investigation by the Federal Energy Regulatory Commission. The dataset was purchased by Leslie Kaelbling but found to have integrity problems. These problems were fixed by folks at SRI which made the corpus usable. It is now distributed by William Cohen at:

http : //www.cs.cmu.edu/ enron/

18 Source Code

18.1 Feature Scripts

18.1.1 CentroidSimLocal.pl

```
#!/cs/public/bin/perl
#
# usage: echo cluster_file_name | Centroid_Sim_Local.pl <idffile> <datadir>
#

#
# <idffile> is relative to <datadir>
#

use strict;

use FindBin;
use lib "$FindBin::Bin/../../lib/", "$FindBin::Bin/../../lib/arch/";

use Essence::IDF;
use Essence::Centroid;

use MEAD::SentFeature;

# command-line args.
my $idf_file = shift;
my $datadir = shift;

# open the specified IDF file.
open_nidf($datadir.'/../'.$idf_file);

# Centroid and the max value for any sentence.
my $centroid = Essence::Centroid->new();
my $max_cent = 0;

extract_sentfeatures($datadir, {'Cluster'=>\&cluster,
                                'Sentence' =>\&sentence});

sub cluster {
    my $cℓuster = shift;
    my $query = shift; # ignored.

    my @sents;

    foreach my $did (keys %{$cℓuster}) {
        my $docref = $$cℓuster{$did};

        my $text;
        foreach my $sentref (@{$docref}) {
            $text .= " " . $$sentref{'TEXT'};
            push @sents, $$sentref{'TEXT'};
        }

        $centroid->add_document($text);
    }

    foreach my $s (@sents) {
        my $score = $centroid->centroid_score($s);
        if ($score > $max_cent) {
            $max_cent = $score;
        }
    }

}

sub sentence {
    my $feature_vector = shift;
    my $attribs = shift;

    my $text = $$attribs{"TEXT"};
    my $score = $centroid->centroid_score($text);
    $$feature_vector{'Centroid'} = sprintf("%17.15f", $score / $max_cent);
}
```

18.1.2 Class.pl

```
#!/cs/public/bin/perl
                                                                          70
#
# usage: echo cluster_file query_file | Class.pl <datadir>
#
# Note: datadir is appended by the driver.pl script as it
# calls the feature script.
#

use strict;

use FindBin;                                                              80
use lib "$FindBin::Bin/../../lib", "$FindBin::Bin/../../lib/arch";

use MEAD::SentFeature;

my $datadir = shift;
my @scores;


        #open GoldStandard file and parse out score
        open(DATA, $datadir."/../goldstandard.csv") or die("Error opening goldstandard: ".$datadir."/../goldstandard.csv"); 90
        <DATA>; #The first line is just labels
        while (<DATA>)
        {
                chomp;
                split(',');
                # 3*essential + optional
                $scores[$_[0]][$_[1]] = 3*$_[2]+$_[3];
        }
        close DATA;
                                                                          100

extract_sentfeatures($datadir, {'Sentence' => \&sentence});

sub sentence {
    my $feature_vector = shift;
    my $sentref = shift;

    my $did = $$sentref{"DID"};
    my $sno = $$sentref{"SNO"};
    my $text = $$sentref{"TEXT"};                                         110

    # You can compute more than one feature at a time,
    # but all but one may be "lost" as driver.pl looks for features
    # in files with names that include the feature name.
    $$feature_vector{"Class"} = $scores[$did][$sno];
    # etc...
}
```

18.1.3 IsQuestion.pl

```
#!/cs/public/bin/perl

#                                                                         120
# usage: echo cluster_file query_file | Is_Question.pl <datadir>
#
# Note: datadir is appended by the driver.pl script as it
# calls the feature script.
#

use strict;

use FindBin;
use lib "$FindBin::Bin/../../lib", "$FindBin::Bin/../../lib/arch";        130

use MEAD::SentFeature;

my $datadir = shift;

extract_sentfeatures($datadir, {'Cluster' => \&cluster,
                                'Document' => \&document,
                                'Sentence' => \&sentence});

sub cluster {                                                             140
    my $clusterref = shift;
    my $queryref = shift;          # This will only be defined if a
                                   # query filename is passed via the
                                   # standard input along with the
                                   # cluster filename.
```

```
        foreach my $did (keys %$clusterref) {
                # This cycling through the DIDs will produce each
                # document passed to the document subroutine.
                my $docref = $$clusterref{$did};                                150

        }
}

sub document {
        my $docref = shift;

        for my $sno ( 1 .. (scalar(@$docref)-1) ) {
                # This will produce each sentence in the document,
                # and because the document subroutine is called with             160
                # each document, it will produce every sentence in the
                # cluster.
                my $sentref = $$docref[$sno]; # note array indices, not hash keys.

        }
}

sub sentence {
        my $feature_vector = shift;
        my $sentref = shift;                                                    170

        my $did = $$sentref{"DID"};
        my $sno = $$sentref{"SNO"};
        my $text = $$sentref{"TEXT"};

        # You can compute more than one feature at a time,
        # but all but one may be "lost" as driver.pl looks for features
        # in files with names that include the feature name.

                #Does the sentence contain a question mark?                      180
        if ($text =~ m/\?/)
        {
                $$feature_vector{"Is_Question"} = "1";
        }
        else
        {
                $$feature_vector{"Is_Question"} = "0" ;
        }
        # etc...
}                                                                               190
```

18.1.4 MRelPos.pl

```
#!/cs/public/bin/perl

#
# usage: echo cluster_file query_file | M_Rel_Pos.pl <datadir>
#
# Note: datadir is appended by the driver.pl script as it
# calls the feature script.
#

use strict;                                                                     200

use FindBin;
use lib "$FindBin::Bin/../../lib", "$FindBin::Bin/../../lib/arch";

use MEAD::SentFeature;

my $datadir = shift;
my $tsent; #count total number of sentences in message

extract_sentfeatures($datadir, {'Cluster' => \&cluster,                         210
                                'Document' => \&document,
                                'Sentence' => \&sentence});

sub cluster {
        my $clusterref = shift;
        my $queryref = shift;           # This will only be defined if a
                                        # query filename is passed via the
                                        # standard input along with the
                                        # cluster filename.
                                                                                220
        foreach my $did (keys %$clusterref) {
                # This cycling through the DIDs will produce each
                # document passed to the document subroutine.
```

```
        my $docref = $$cℓusterref{$did};

    }
}

sub document {
    my $docref = shift;                                                    230

    for my $sno ( 1 .. (scalar(@$docref)−1) ) {
        # This will produce each sentence in the document,
        # and because the document subroutine is called with
        # each document, it will produce every sentence in the
        # cluster.
        my $sentref = $$docref[$sno]; # note array indices, not hash keys.

    }
        $tsent = scalar(@$docref)−1;                                       240
}

sub sentence {
    my $feature_vector = shift;
    my $sentref = shift;

    my $did = $$sentref{"DID"};
    my $sno = $$sentref{"SNO"};
    my $text = $$sentref{"TEXT"};
                                                                           250
    # You can compute more than one feature at a time,
    # but all but one may be “lost” as driver.pl looks for features
    # in files with names that include the feature name.
    $$feature_vector{"M_Rel_Pos"} = ($sno−1)/$tsent;
    # etc...
}
```

18.1.5 MsgNum.pl

```
#!/cs/public/bin/perl

#
# usage: echo cluster_file query_file | Msg_Num.pl <datadir>              260
#
# Note: datadir is appended by the driver.pl script as it
# calls the feature script.
#

use strict;

use FindBin;
use lib "$FindBin::Bin/../../lib", "$FindBin::Bin/../../lib/arch";
                                                                           270
use MEAD::SentFeature;

my $datadir = shift;

extract_sentfeatures($datadir, {'Cluster' => \&cluster,
                                'Document' => \&document,
                                'Sentence' => \&sentence});

sub cluster {
    my $cℓusterref = shift;                                                280
    my $queryref = shift;           # This will only be defined if a
                                    # query filename is passed via the
                                    # standard input along with the
                                    # cluster filename.

    foreach my $did (keys %$cℓusterref) {
        # This cycling through the DIDs will produce each
        # document passed to the document subroutine.
        my $docref = $$cℓusterref{$did};
                                                                           290
    }
}

sub document {
    my $docref = shift;

    for my $sno ( 1 .. (scalar(@$docref)−1) ) {
        # This will produce each sentence in the document,
        # and because the document subroutine is called with
        # each document, it will produce every sentence in the             300
        # cluster.
```

```
        my $sentref = $$docref[$sno]; # note array indices, not hash keys.

    }
}

sub sentence {
    my $feature_vector = shift;
    my $sentref = shift;
                                                                        310
    my $did = $$sentref{"DID"};
    my $sno = $$sentref{"SNO"};
    my $text = $$sentref{"TEXT"};

    # You can compute more than one feature at a time,
    # but all but one may be "lost" as driver.pl looks for features
    # in files with names that include the feature name.

    $$feature_vector{"Msg_Num"} = $did;
    # etc...                                                            320
}
```

18.1.6 Tfidfavg.pl

```
#!/cs/public/bin/perl

#
# usage: echo cluster_file query_file | Tfidfavg.pl <idffile> <datadir>
#
# Note: datadir is appended by the driver.pl script as it
# calls the feature script.
#
                                                                        330
use strict;

use FindBin;
use lib "$FindBin::Bin/../../lib", "$FindBin::Bin/../../lib/arch";

use Essence::IDF;
use Essence::Text;
use MEAD::SentFeature;

my $idffiℓe = shift;                                                    340
my $datadir = shift;

open_nidf($idffiℓe);

extract_sentfeatures($datadir, {'Cluster' => \&cluster,
                                'Document' => \&document,
                                'Sentence' => \&sentence});

sub cluster {
    my $cℓusterref = shift;                                             350
    my $queryref = shift;            # This will only be defined if a
                                     # query filename is passed via the
                                     # standard input along with the
                                     # cluster filename.

    foreach my $did (keys %$cℓusterref) {
        # This cycling through the DIDs will produce each
        # document passed to the document subroutine.
        my $docref = $$cℓusterref{$did};
                                                                        360
    }
}

sub document {
    my $docref = shift;

    for my $sno ( 1 .. (scalar(@$docref)−1) ) {
        # This will produce each sentence in the document,
        # and because the document subroutine is called with
        # each document, it will produce every sentence in the          370
        # cluster.
        my $sentref = $$docref[$sno]; # note array indices, not hash keys.

    }
}

sub sentence {
    my $feature_vector = shift;
    my $sentref = shift;
```

```
                                                                                    380
        my  $did  =  $$sentref{"DID"};
        my  $sno  =  $$sentref{"SNO"};
        my  $text  =  $$sentref{"TEXT"};

        #  You  can  compute  more  than  one  feature  at  a  time,
        #  but  all  but  one  may  be  "lost"  as  driver.pl  looks  for  features
        #  in  files  with  names  that  include  the  feature  name.
                my  %hash;

                my  @words  =  split_words($text);                                   390
                foreach  my  $word  (@words)  {
                        $hash{$word}++;
                }
                my  $totaℓ  =  0;
                while(  (my  $word,  my  $tf)  =  each  %hash)
                {
                                #There  is  a  difference  in  Clairlib  word-splitting  and  MEAD  word-splitting.  The  IDF  database  was  built  with  Clairlib,  while  this  funct
                                my  $formatted_word  =  lc  $word;
                                $formatted_word  =~  s/\'/~/;  #replace  '  with  ~
                                $formatted_word  =~  s/\/|\\|"//;  #remove  quoatations  and  slashes      400
                                $formatted_word =~ s/(\.|;)$//; #remove period from last word
                                $total += $tf/@words * get_nidf($formatted_word);
                }
        $$feature_vector{"Tfidfavg"} = $total / @words;
        # etc...
}
```

18.1.7 Tfidfsum.pl

```
#!/cs/public/bin/perl

#
#  usage:  echo  cluster˙file  query˙file  —  Tfidfsum.pl  ¡idffile¿  ¡datadir¿          410
#
#  Note:  datadir  is  appended  by  the  driver.pl  script  as  it
#  calls  the  feature  script.
#

use  strict;

use  FindBin;
use  lib  "$FindBin::Bin/../../lib",  "$FindBin::Bin/../../lib/arch";
                                                                                    420
use  Essence::IDF;
use  Essence::Text;
use  MEAD::SentFeature;

my  $idffiℓe  =  shift;
my  $datadir  =  shift;

open_nidf($idffiℓe);

extract_sentfeatures($datadir,  {'Cluster'  =>  \&cluster,                          430
                                                      'Document'  =>  \&document,
                                                      'Sentence'  =>  \&sentence});

sub  cluster  {
        my  $cℓusterref  =  shift;
        my  $queryref  =  shift;                       #  This  will  only  be  defined  if  a
                                                       #  query  filename  is  passed  via  the
                                                       #  standard  input  along  with  the
                                                       #  cluster  filename.
                                                                                    440
        foreach  my  $did  (keys  %$cℓusterref)  {
                #  This  cycling  through  the  DIDs  will  produce  each
                #  document  passed  to  the  document  subroutine.
                my  $docref  =  $$cℓusterref{$did};

        }
}

sub  document  {
        my  $docref  =  shift;                                                      450

        for  my  $sno  (  1  ..  (scalar(@$docref)−1)  )  {
                #  This  will  produce  each  sentence  in  the  document,
                #  and  because  the  document  subroutine  is  called  with
                #  each  document,  it  will  produce  every  sentence  in  the
                #  cluster.
                my  $sentref  =  $$docref[$sno];  #  note  array  indices,  not  hash  keys.
```

```
        }
}                                                                          460

sub sentence {
        my $feature_vector = shift;
        my $sentref = shift;

        my $did = $$sentref{"DID"};
        my $sno = $$sentref{"SNO"};
        my $text = $$sentref{"TEXT"};

        # You can compute more than one feature at a time,                  470
        # but all but one may be "lost" as driver.pl looks for features
        # in files with names that include the feature name.
                my %hash;

                my @words = split_words($text);
                foreach my $word (@words) {
                        $hash{$word}++;
                }
                my $total = 0;
                while( (my $word, my $tf) = each %hash)                    480
                {
                                #There is a difference in Clairlib word-splitting and MEAD word-splitting. The IDF database was built with Clairlib, while this funct
                                my $formatted_word = lc $word;
                                $formatted_word =~ s/\'/~/; #replace ' with ~
                                $formatted_word =~ s/\/|\\|"//; #remove quoatations and slashes
                                $formatted_word =~ s/(\.|;)$//; #remove period from last word
                                $total += $tf/@words * get_nidf($formatted_word);
                }
        $$feature_vector{"Tfidfsum"} = $total;
        # etc...                                                            490
}
```

18.1.8 ThreadLineNum.pl

```
#!/cs/public/bin/perl

#
# usage: echo cluster_file query_file | Thread_Line_Num.pl <datadir>
#
# Note: datadir is appended by the driver.pl script as it
# calls the feature script.
#
                                                                           500
use strict;

use FindBin;
use lib "$FindBin::Bin/../../lib", "$FindBin::Bin/../../lib/arch";

use MEAD::SentFeature;

my $datadir = shift;
my @maxsent; #count number of sentences in document
                                                                           510
extract_sentfeatures($datadir, {'Cluster' => \&cluster,
                                'Document' => \&document,
                                'Sentence' => \&sentence});

sub cluster {
        my $clusterref = shift;
        my $queryref = shift;           # This will only be defined if a
                                        # query filename is passed via the
                                        # standard input along with the
                                        # cluster filename.                 520

        foreach my $did (keys %$clusterref) {
                # This cycling through the DIDs will produce each
                # document passed to the document subroutine.
                my $docref = $$clusterref{$did};

                $maxsent[$did] = scalar(@$docref)-1;
        }
}
                                                                           530
sub document {
        my $docref = shift;

        for my $sno ( 1 .. (scalar(@$docref)-1) ) {
                # This will produce each sentence in the document,
```

```
            # and because the document subroutine is called with
            # each document, it will produce every sentence in the
            # cluster.
            my $sentref = $$docref[$sno]; # note array indices, not hash keys.
                                                                                540
        }
}

sub sentence {
        my $feature_vector = shift;
        my $sentref = shift;

        my $did = $$sentref{"DID"};
        my $sno = $$sentref{"SNO"};
        my $text = $$sentref{"TEXT"};                                           550

        # You can compute more than one feature at a time,
        # but all but one may be "lost" as driver.pl looks for features
        # in files with names that include the feature name.
        my $totsent = 0; #total number of sentences
        for my $i ( 0 .. (scalar($did)-1) ) {
            #find number of sentences in the thread before this thread
            $totsent += $maxsent[$i];
        }
        $$feature_vector{"Thread_Line_Num"} = $sno+$totsent;                    560
        # etc...
}
```

18.1.9 TRelPos.pl

```
#!/cs/public/bin/perl

#
# usage: echo cluster_file query_file | T_Rel_Pos.pl <datadir>
#
# Note: datadir is appended by the driver.pl script as it
# calls the feature script.
#                                                                               570

use strict;

use FindBin;
use lib "$FindBin::Bin/../../lib", "$FindBin::Bin/../../lib/arch";

use MEAD::SentFeature;

my $datadir = shift;
my @maxsent; #count number of sentences in document                            580
my $tsent; #count total number of sentences in thread

extract_sentfeatures($datadir, {'Cluster' => \&cluster,
                                'Document' => \&document,
                                'Sentence' => \&sentence});

sub cluster {
        my $clusterref = shift;
        my $queryref = shift;           # This will only be defined if a
                                        # query filename is passed via the      590
                                        # standard input along with the
                                        # cluster filename.

        foreach my $did (keys %$clusterref) {
            # This cycling through the DIDs will produce each
            # document passed to the document subroutine.
            my $docref = $$clusterref{$did};

            $maxsent[$did] = scalar(@$docref)-1;
            $tsent += scalar(@$docref)-1;                                       600
        }
}

sub document {
        my $docref = shift;

        for my $sno ( 1 .. (scalar(@$docref)-1) ) {
            # This will produce each sentence in the document,
            # and because the document subroutine is called with
            # each document, it will produce every sentence in the              610
            # cluster.
            my $sentref = $$docref[$sno]; # note array indices, not hash keys.
```

```
        }
}

sub sentence {
        my $feature_vector = shift;
        my $sentref = shift;
                                                                                620
        my $did = $$sentref{"DID"};
        my $sno = $$sentref{"SNO"};
        my $text = $$sentref{"TEXT"};

        # You can compute more than one feature at a time,
        # but all but one may be "lost" as driver.pl looks for features
        # in files with names that include the feature name.
        my $totsent = 0; #total number of sentences
        for my $i ( 0 .. (scalar($did)-1) ) {
#find the number of sentences in the message before this message                630
                $totsent += $maxsent[$i];
        }
                #Computes the ratio of the number of sentences preceding this one divided by the total number of sentences in the thread
        $$feature_vector{"T_Rel_Pos"} = ($sno+$totsent-1)/$tsent;
        # etc...
}
```

18.2 Classifier

18.2.1 weka-classifier.pl

```
#!/cs/public/bin/perl
#This is the classifier that calls the weka model
#usage: weka-classifier.pl <classifier> <classifier-model> ...
                                                                                640
use FindBin;
use lib "$FindBin::Bin/../lib", "$FindBin::Bin/../lib/arch";

use XML::Parser;
use XML::Writer;

use MEAD::MEAD;
use MEAD::Cluster;
use MEAD::SentFeature;
                                                                                650
my %fnames = ();
my @all_sents = ();
my %features = ();
my $classifier;
my $model;

{
        &parse_options();

        %features = read_sentfeature();                                         660
        @all_sents = @{ flatten_cluster(\%features) };

        ##Score the sentences based upon their weights
        &compute_scores();

        ##Write out the new scores to a sentjudge file
        &write_sentjudge();
}

sub compute_scores {                                                            670
                #Create Arff file
                write_features();
                my $command = "java -Xmx1000m -classpath /cs/public/lib/weka-3-5-6/weka.jar $classifier -T sum.arff -l $model -c
                open RES, $command;
                my $t = <RES>;
                ($t eq "predicted\n") or die ("WEKA provided bad output: ".$t."\n");
                my @res = ();
                while(<RES>)
                {
                                chomp;                                          680
                                push @res, $_;
                }
                close FILE;
        ##Now compute sentence scores based on weight
                foreach my $sentref (@all_sents)
                {
                                $$sentref{'FinalScore'} = shift @res;
                }
```

```
                                                                              690
}

sub write_features {

        my $sentfeatures = \%features;
        open(OUT, ">sum.arff");
        # get the feature names to print.
        my @dids = keys %$sentfeatures;
        my $did1 = $dids[0];
        my $sentref = $$sentfeatures{$did1}[1];                               700
        my @feature_names = keys %$sentref;
         my $did_format = '%-' . "$longest.$longest" . 's';
        # print the feature names
        print OUT '@relation '."'summary'\n";

        foreach my $fn (@feature_names) {
                print OUT '@attribute '.$fn." real\n";
        }
         print OUT '@data'."\n";
                                                                              710
        foreach my $did (keys %$sentfeatures) {

                my $fdocref = $$sentfeatures{$did};
                for (my $sno = 1; $sno < @$fdocref; $sno++) {

                        my $fsentref = $$fdocref[$sno];
                        my $first;
                        foreach my $fn (@feature_names) {
                                if (defined($first)) {print OUT ","};
                                print OUT $$fsentref{$fn};                    720
                                $first = 1;
                        }

                        print OUT "\n";
                }
        }
        close OUT;
}
sub parse_options {
                                                                              730
    ##If there's no input file
    if (@ARGV < 2 ) {
        Debug ("usage: weka-classifier.pl <classifier> <classifier-model> ..." , 3, "ParseOpts");
        exit(1);
    }
        $classifier = shift @ARGV;
        $model = shift @ARGV;
}

sub write_sentjudge {                                                         740

    my $writer = new XML::Writer(DATA_MODE=>1);

    $writer->xmlDecl();
    $writer->doctype("SENT-JUDGE", "", "/clair/tools/mead/dtd/sentjudge.dtd");

    $writer->startTag("SENT-JUDGE", "QID"=>"none");

    ##MEAD input doesn't store RSNT and PAR.          For now, don't output it
    foreach my $sentref (@all_sents) {                                        750
        $writer->startTag("S",
                          "DID"=>$$sentref{"DID"},
                          "SNO"=>$$sentref{"SNO"});

        $writer->emptyTag("JUDGE",
                          "N" => "CLASSIFIER",
                          "UTIL"=>$$sentref{"FinalScore"});
        $writer->endTag();
    }
                                                                              760
    $writer->endTag();
    $writer->end();
}
```

18.2.2 makearff.pl

```perl
#This file will parse a mead.pl -score output into an arff file

my $out = '@relation \'summary\''."\n"; #output file string

#The first line is the header
my @labels = split(/\s+/,<>);
for (2..$#labels-1)
{
	$out .= '@attribute '.$labels[$_]." real\n";
}
#The rest of the file is data
$out .= '@data'."\n";
while(<>)
{
	my @data = split(/\s+/,$_);
	my $s = join(",",@data[2..$#data-1]);
	$out .= $s."\n";
}
print $out;
```

18.2.3 createallarff.sh

```bash
#!/bin/bash
#This is a bash script that calls make_arff on all threads
for i in `ls /var/tmp/research/data`; do
perl ~/mead/bin/mead.pl -scores $i | perl ~/mead/user/mead/bin/make_arff.pl > /var/tmp/research/data/$i/features.arff;
done
```

18.2.4 meadrc

```
compression_basis	sentences
compression_percent	30
#output_mode	scores
output_mode	summary
#system	RANDOM

data_path	/var/tmp/research/data
#data_path	/.autofs/homes/ubccshome/u/ulrichj/research/data

#Basic features
feature Thread_Line_Num ~/mead/user/mead/bin/feature-scripts/Thread_Line_Num.pl
feature Centroid_Sim ~/mead/bin/feature-scripts/Centroid.pl ~/mead/etc/enidf ENG
feature Centroid_Sim_Local ~/mead/user/mead/bin/feature-scripts/Centroid_Sim_Local.pl myidf
feature Length ~/mead/bin/feature-scripts/Length.pl
feature Tfidfsum ~/mead/user/mead/bin/feature-scripts/Tfidfsum.pl enronidf-punc
feature Tfidfavg ~/mead/user/mead/bin/feature-scripts/Tfidfavg.pl enronidf-punc
feature T_Rel_Pos ~/mead/user/mead/bin/feature-scripts/T_Rel_Pos.pl
feature Is_Question ~/mead/user/mead/bin/feature-scripts/Is_Question.pl
#Basic+ features
feature Msg_Num ~/mead/user/mead/bin/feature-scripts/Msg_Num.pl
feature M_Rel_Pos ~/mead/user/mead/bin/feature-scripts/M_Rel_Pos.pl
#Gold Standard Rating
feature Class ~/mead/user/mead/bin/feature-scripts/Class.pl

#classifier ~/mead/user/mead/bin/weka-classifier.pl weka.classifiers.functions.SMOreg /var/tmp/research/cross-validation/model
```

18.2.5 combinearff.pl

```perl
#This file is used to take the 39 thread .arffs and divide
#them into the training and testing arff depending on the
#ordering file
$filename = shift;
my $data;
my $header;
my $data2;
my $header2;
my $count = 0;

open(NAMES, $filename) or die("Can't open ".$filename);
while(<NAMES>)
{
	chomp;
	open(ARFFFILE, "/var/tmp/research/data/".$_."/features.arff") or die("Can't find file: "."/var/tmp/research/data/".$_."/f
	my $datastart = 0;
	if ($count > 3)
	{
		$header = "";
	}
	else
	{
```

```
                $header2 = "";
        }
        while($ℓ = <ARFFFILE>)
        {
                if ($datastart)
                {                                                                   840
                        if ($count > 3)
                        {
                        $data .= $ℓ;
                        }
                        else
                        {
                        $data2 .= $ℓ;
                        }
                }
                else                                                                850
                {
                        if ($count > 3)
                        {
                        $header .= $ℓ;
                        }
                        else
                        {
                        $header2 .= $ℓ;
                        }
                }                                                                   860
                if ($ℓ =~ /\@data/)
                {
                        $datastart = 1;
                }
        }
        close ARFFFILE;
        $count++;
}
close NAMES;
open(TRAIN, ">arff/".$fiℓename.".train.arff") or die("Can't open output file");    870
open(TEST, ">arff/".$fiℓename.".test.arff") or die("Can't open output file");
print TRAIN $header;
print TRAIN $data;
print TEST $header2;
print TEST $data2;
close TRAIN;
close TEST;
```

18.2.6 createarff.sh

```
#!/bin/bash
#This script creates 10 training and testing arffs                                 880
for i in        1 2 3 4 5 6 7 8 9 10; do
perl combine_arff.pl $i;
done
```

18.2.7 createmodels.sh

```
#!/bin/bash
#This script creates the classifier model from the trainig.arff
for i in 1 2 3 4 5 6 7 8 9 10; do
java −Xmx1000m −classpath /cs/public/lib/weka−3−5−6/weka.jar weka.classifiers.meta.Bagging −t arff/$i.train.arff −d models/$i.Bagging −x 0 −c 9;
done
```

18.2.8 files.sh

```
#!/bin/bash
#This script lists the different thread names                                      890
for i in ‘ls ../data‘; do
echo $i;
done
```

18.2.9 randomize.pl

```
#!/cs/public/bin/perl -w
#Fisher Yates Shuffle algorithm
#This creates the random 10 distributions for the 10 fold cross-validation
sub fys {
my $array=shift;
my $i;
for ( $i=@$array; −−$i; ) {                                                         900
my $j = int rand ($i+1);
next if $i == $j;
@$array[$i,$j]=@$array[$j,$i];
}
```

```
}

while (<>) {
push(@lines, $_);
}
fys(\@lines);                                                        910
foreach(@lines) {
print $_;
}
```

18.3 Create IDF

18.3.1 dirtocor.sh

```
#!/bin/bash
#For all threads add them to clairlib corpus
for i in `ls orig`; do
~/clairlib−core/util/directory_to_corpus.pl −b produced −c $i −d orig/$i;
done
```

18.3.2 idfquery.sh

```
#!/bin/bash                                                          920
#Create individual idf file for each thread
for i in `ls orig`; do
~/clairlib−core/util/idf_query.pl −b produced −c $i −−all > data/$i/myidf.txt;
done
```

18.3.3 indcor.sh

```
#!/bin/bash
#For each thread index the corpus
for i in `ls orig`; do
~/clairlib−core/util/index_corpus.pl −b produced −c $i;
done
```

18.3.4 mkdb.sh

```
#!/bin/bash                                                          930
#Make a db file for each thread
for i in `ls orig`; do
awk '{ gsub(/: /, " "); print }' data/$i/myidf.txt > data/$i/myidf1.txt;
mv −f data/$i/myidf1.txt data/$i/myidf.txt;
~/mead/bin/write−idf.pl data/$i/myidf data/$i/myidf.txt;
done
```

18.4 Generate Summaries

18.4.1 createsummaries.sh

```
#!/bin/bash
#Generate summaries from mead for each fold
for i in 1 2 3 4 5 6 7 8 9 10; do
count=0;                                                             940
for j in `cat /var/tmp/research/cross−validation/$i`; do
if [[ $count −lt 4 ]]; then
~/mead/bin/mead.pl −classifier "~/mead/user/mead/bin/weka-classifier.pl weka.classifiers.meta.Bagging /var/tmp/research/cross-valida
fi
count=`expr $count + 1`;
done
done
```

18.4.2 sumremnum.sh

```
#!/bin/bash
#This script removes the line number from the MEAD summaries
for i in `ls summaries`; do                                          950
cat summaries/$i | awk '{ sub(/\[[[:digit:]]+\][[:space:]]*/,"");     print }' > summaries/$i;
done
```

18.5 Rouge Evaluation

18.5.1 clueawk.sh

```
#!/bin/bash
#Create ROUGE scores for cluescore summaries
./create_lst.sh | awk −F. '{print "/var/tmp/research/data/" $2 "/sum.k30.n100 /var/tmp/research/data/" $2 "/GS8.txt"}'
```

18.5.2 myawk.sh

```
#!/bin/bash
#Create ROUGE scores for summaries from create_lst.sh
./create_lst.sh | awk -F. '{print $1 "." $2 "." $3 "." $4 " /var/tmp/research/data/" $2 "/GS8.txt"}'
```

18.5.3 createlst.sh

```
#!/bin/bash                                                                                      960
#Create ROUGE scores for machine learning summaries
for i in `ls /var/tmp/research/cross-validation/summaries/*.SMOreg`; do
echo $i;
done
```

18.6 Mean Absolute Error

18.6.1 compileerr.sh

```
#!/bin/bash
#Extract Mean Absolute Error from weka output
for i in 1 2 3 4 5 6 7 8 9 10; do
./meanabserr.sh $i meta.Bagging | grep "Mean absolute" | awk 'NR>1 {print $4}';
done
```

18.6.2 meanabserr.sh

```
#!/bin/bash                                                                                      970
#Run weka in cross-validation to generate results
java -Xmx1000m -classpath /cs/public/lib/weka-3-5-6/weka.jar weka.classifiers.$2 -t arff/$1.train.arff -d model -x 0 -c 9;
echo "Created Model";
java -Xmx1000m -classpath /cs/public/lib/weka-3-5-6/weka.jar weka.classifiers.$2 -T arff/$1.test.arff -l model -c 9;
```