SentiSummary: Sentiment Summarization for User Product Reviews

Dept. of CIS - Senior Design 2009-2010

Matthew Feczko
feczko@seas.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

Andrew Schaye
schayeaj@seas
Univ. of Pennsylvania
Philadelphia, PA

M. Marcus & A. Nenkova
{mitch, nenkova}@seas
Univ. of Pennsylvania
Philadelphia, PA

ABSTRACT

This paper describes an approach to summarize and analyze user product reviews using summarization techniques found in multi-document summarization as well as sentiment analysis to extract opinions found in text. Previously, most research has focused on creating summaries for formal text such as The Wall Street Journal or other major news outlet. However, those same summarizers do not perform well on informal language such as blogs and user reviews. Our system uses sentiment analysis to analyze user product reviews for multiple products, identify and parse the positive and negative viewpoints across multiple reviews, summarize those viewpoints and display the aggregated information in a user friendly and useful manner.

1. INTRODUCTION

From product reviews to opinion editorials, the internet is now our main avenue for researching, purchasing and finding information. We locate various forms of information scattered across billions of webpages including thousands of blogs, Twitter pages, and Amazon product reviews to name a few. With so many resources for general product information, detailed technical specifications, and thousands of professional and user reviews, the amount of data can be quite overwhelming. Consider a product on Amazon, say a third generation, 64 gigabyte Apple iPod Touch: there are over 1200 reviews1. No one has the time to read all of these reviews and aggregate all the information and opinions contained within them. It would be much easier to read a summarized version that captures most of the opinions found within those reviews rather than having to read through all 1200 reviews. A better way to research a paper or digest daily news and media would be through some form of categorization and summarization all of these data. That summarization task can save us countless hours by separating out the important from unimportant information and yielding a smaller, more simplified version of what we are viewing.

While current methods of statistical summarization work well with traditional media, such as news articles from The Wall Street Journal or academic papers from the Association for Computational Linguistics2, those same methods do not work well for most blogs, opinion pieces, and product reviews, from sources like Engadget3 or Amazon. The main reason that current methods do not work well is that they are variations and combinations of one or more statistical methods. They fail to take into account sentiment, or the emotions and opinions of the author.

Over the last decade, there have been many advances in sentiment analysis and opinion mining. The work of Kim et al. [4] and Lerman et al. [6] have demonstrated improvements and new methods for summarizing opinion media where straight statistical summarization has historically failed. Moreover, Lerman et al. shows that not only do users prefer summarizers that are based on a sentiment model, but that sentiment-based summarizers can, in many cases, reduce error and create more meaningful, opinion-based summaries.

As will be explained in great detail below, SentiSummary uses a multi-step methodology that combines cutting edge techniques in multi-document summarization, polarity classification and sentiment analysis. The goal of SentiSummary is to automatically synthesize, filter, and summarize user product reviews into a short list of positive and negative opinions as expressed in the reviews.

2. RELATED WORK

The nature of our project requires us to combine many areas of past and current natural language processing research. SentiSummary covers multi-document summarization, part-of-speech tagging, sentiment analysis, machine learning classification, and evaluation of summarization systems. Although none of these areas are new to natural language processing, as each area reaches new levels of proficiency and approach replicating human summarization and language models, the way in which we combine these research topics provides better methodologies for summarization and sentiment analysis.

One of our major tasks in SentiSummary is multi-document summarization of many different reviews. We will base much of our work on Schiffman et al.’s “Experiments in Multi-Document Summarization” [15]. The goal of Schiffman et al.’s research is to use new methods for finding “interesting and informative sentences” to enhance multi-document summarization. Their system uses proven statistical methods, such as word and concept frequencies, as well as many other heuristics to accomplish two major goals: first, rank sentences in order to minimize overlapping and second, maxi-

1As of 18 April 2010 at 3:45PM EST
2http://www.aclweb.org/
3http://www.engadget.com/
mize the information extracted from the documents. We will use this basic framework with additional statistical measures in conjunction with our sentiment model to produce the final positive and negative opinion lists.

Much research has been done on Part-of-Speech tagging (POS tagging) since the 1960s [9]. Linguists and computer scientists alike have worked on using supervised and unsupervised approaches to tagging the parts of speech, using Hidden Markov Models and Bayesian Logic as well as similarly pre-tagged data. In 1993, Marcus completed the Penn Treebank Project [10] which annotated naturally occurring text for linguistic structure and parts of speech from the WSJ. In a more recent article, Zhao and Marcus [17] propose a new model for unsupervised POS tagging which requires far less information and simpler computation methods. Their model uses linguistic distinctions between open and closed-class items. In other words, the model focuses on the information gained from those word types that are open (i.e. words can be added to the type at any time) versus those types that have a fixed, or closed, set of words. Though not as sophisticated as fully supervised systems, Marcus’s approach requires very little existing information to effectively perform POS tagging.

A combination of topic words and other statistically relevant words will be used to help cluster words into specific groupings. SentiSummary will use two levels of clustering. Layer one uses a classification algorithm to create two “bags” of positive and negative sentences, while layer two will cluster similar sentences within each “bag” around specific features of a product or specific ideas expressed. It should be noted that due to time restrictions and lack of data, the current version of SentiSummary does not include this second layer of clustering.

Liu et al.’s research on event term clustering for use in extractive summarization [8] will provide the primary framework for our topic-based clustering algorithm. They use VerbOcean [1], a verb relation repository with fine grained relations, to calculate a neighborhood density for a given term. The highest density neighborhoods are considered clusters on a given topic. This will help us group our chunks into appropriate topics for our summaries. We will also use the cognitive synonyms (synsets) of WordNet [11], developed at Princeton University, to add conceptual-semantic and lexical relations, such as synonyms, antonyms, etc. These will be included in later versions of SentiSummary.

Classification of positive versus negative is the crucial piece of SentiSummary. There are many classification models out there that have their pros and cons. Some of the original models were based on perceptron and voted perceptron hyperplane models. These models take pre-classified data and attempt to build a separating hyperplane to help classify. Support Vector Machines are the more advanced version of the perceptron models. The concepts behind these are described by Sassi for Japanese Analysis Tasks [14], which are similar to SentiSummary’s classification algorithms. A second set of classification types, of which the above are a subset, are feature vector classification algorithms. Naive Bayes, Stepwise Regression, and Ridge Regression are just a few examples. In Veeramachaneni and Kondadadi’s work on Surrogate Learning [16], they explain about a semi-supervised feature vector classification model that is very similar to ours. SentiSummary uses Ridge Regression as it is most accurate given our training data, which we describe in more detail later.

Because our sentiment model is based heavily on SentiWordNet (see more information in Section: 3.2.3 or the original paper [2]), misspellings have the potential to severely hurt our sentiment scoring algorithm. At first, SentiSummary attempted to use a basic spell checker based on Ku- kich’s work [5]. Kukich introduced the problem of spelling correction as a progressively harder problem that starts with nonword error detection, isolated-word error correction and finally, context-dependent word correction. As Kukich explains, there are many different types of errors, such as insertion, deletion, substitution and transpositions, as well as many ways to correct to these errors—for example, minimum edit distance, similar key, n-gram, and rule-based techniques. Because SentiSummary uses a vector-based algorithm in which sentiment is a few of the features, spelling mistakes lacked to influence that they previously held when SentiSummary was attempting to score sentences based on SentiWordNet alone, and thus why SentiSummary no longer includes spell checking. Although spell checking may improve the accuracy of SentiSummary as a whole, other aspects were more important in version 1.

As statistical summarization tries to eke out the last few accuracy percentage points, the importance of sentiment continues to grow for a wide range of applications. The four articles discussed below summarize sentiments important [6], ways to measure sentiment based on simple polarity (positive, negative, and objective) [4] and [13], and a parse-and-paraphrase paradigm that parses the adverb-adjective-noun phrases and uses them as clusters to reduce noise [7]. The research of Lerman et al. shows that users have a preference for summarizers based on sentiment models over non-sentiment models, but certain domains (or various categories or subjects) may prefer different models, or ways of scoring sentiment. In our project, we will train on product reviews, as we are building a product review summarizer.

Certain sentiment models may also prove more successful than others. In August 2009, two researchers showed two different sentiment models and explained how they were more successful than other models. Kim et al.’s sentiment model [4] utilizes term weights for sentiment analysis based on collection statistics, contextual and topic-related characteristics, as well as opinion-related properties. These terms weight range from $(-1,1)$ showing a degree of positive vs. negative polarity. Pang and Lee’s methods, as described in their paper [13], are used to decide document-level polarity classification. Applying machine-learning methods, Pang and Lee first decide whether or not a sentence is objective, and then only on subjective sentences (those expressing some opinion) decide the positive/negative classification. We follow a similar method by classifying our chunks (often sentences) with a positive or negative polarity classification.

Liu and Senell’s Parse-and-Paraphrase Paradigm argues that binary polarities may not be as effective when confronted with negation, and the extraction of lexical features into unigrams, bigrams, and N-grams can complicate sentiment with regard to long sentences and implicit long-distance dependency [7]. Liu and Senell propose an approach to extract adverb-adjective-noun phrases based on clause structure in order to accurately score positive vs. negative phrases.

The final area of major research is that of evaluation. We
will be using Nenkova’s “Summarization Evaluation for Text and Speech: Issues and Approach” [12] as the basis of our own evaluation. This paper explains multiple methods of evaluation and describes the advantages and disadvantages of each. Although the focus of this piece is constructing a method for speech summarization, many of the same techniques have been and are used for text summarization evaluation. Although this is central to building our project, evaluation is critical to understanding the significance and value of our work as well as establishing a baseline from which other researchers can compare their algorithms. Evaluation also allows us to evaluate and improve our methods and models in order to create the best overall product. In the current version of SentiSummary, only our classifier has been evaluated, but later versions of SentiSummary will include the above.

3. MODEL, IMPLEMENTATION, AND PERFORMANCE

3.1 System Model
We have developed a tool, SentiSummary, that can automatically summarize product reviews and other opinion material (such as blogs) into a simple positive/negative list about the product. The model incorporates pre-existing natural language processing methods and algorithms as well as our own modules to create a system that analyzes and outputs differently than other current systems. In order to build our sentiment-based automated summarizer, we have used and built modules that recreate other methodologies (that include our own methodology tweaks) necessary to do the statistical and sentimental analysis on top of a framework with which to evaluate our work. The following section outlines the phases of our work and the criteria and methodology of evaluation.

Our system generates a list of positive and negative chunks of specific products extracted from user reviews. Thanks to the help of Mark Dredze, former PhD student at the University of Pennsylvania, we have a sizable corpus of Amazon Product Review data for a multitude of products in various categories (electronics, books, jewelry, etc.). Using this data as our main test corpus, we have created a multi-step process for splicing the user reviews and processing them to extract positive and negative sentences from each review for a given product. Our approach consists of an initialization stage, where we load and train our statistical and sentiment models as well as parse and chunk the reviews, and a processing stage where we evaluate each sentence chunk as a vector of features and build the positive and negative summary lists.

3.2 System Implementation
The following section describes the details of our implementation. Our system is written in Java in a series of classes that represent each of the major modules and structures necessary to best construct our program. We started with Dredze’s data, which had thousands of Amazon reviews in XML files that were divided into categories, such as electronics and books. In order to maximize accuracy and ease of work, we used only the electronics category. Electronics reviews tended to primarily focus on the positive and negative aspects of a product as opposed to a category like movies which tended to consists of summaries. One major downside, however, associated with only using one category is that each product has only a few usable reviews for summarization, as Dredze’s data set rarely included multiple reviews per product.

The entire process described below can either be run from a GUI (Section: 3.2.2) or from a Run class, which outputs to the Console. The GUI easily demonstrates SentiSummary, as users can dynamically view the summaries for a wide range of products instantaneous without requiring re-running the program.

Initially, SentiSummary must load and train the models it will use to process the data for later analysis. Many of these models were pre-built based on the given data in order for us to focus on the summarization aspect as opposed to the data collection. The first model we collected was a statistical model that contains thousands of words linked to their overall frequency and number of documents in which they were found. Extremely common words, such as “and,” “a,” or “the” were removed via a stoplist. These will be used for statistical comparison between sentence chunks as well as part of specific features in our sentence vectors. The next models loaded in are three trained polarity analyzers. Each are nearly identical, except the n-gram models from which they are based. The Polarity Analyzers, based on LingPipe tutorial of Pang and Lee’s work [13], analyzes the chunks for a given polarity. We used a unigram, bigram and trigram models as features in our analysis, which give each chunk a positive or negative for each of the 3 models. The third model is a pre-trained model that loads into the Stanford Log-Linear Part-of-Speech Tagger [3]. The POS tags will be used later in gathering the sentiment values in the vector analysis. Both the POS tagger and Polarity Analyzers are explained in greater detail below in Section 3.2.3. Packages and Pre-built Systems. The final model pre-loaded is our pre-trained Ridge Regression model, which is described in full in the Classifiers Section: 4.1.

Once all of the models have been loaded and trained, we begin the actual processes in SentiSummary. The first step in our processing is to parse the data. The data are located in very large text files of XML that contain multiple parse and syntax errors which continually crashed our parser. XML has a very specific set of keywords that parsers, such as SAX XML parser in SentiSummary, require to be unique to only the XML. In order to parse the XML, the original data files are cleaned, by the FormatFile class, to remove certain phrases that contained “&,” in particular, reviews that contained “&” instead of “and.” These cleaned files were saved to minimize computational time on later runs.

Once cleaned, the Amazon data is parsed out into different aspects of a review by the Parser class. The aspects are saved into a collection of Reviews defined by the Review class. Each Review holds all of the relevant information such as the Product ID, Product Name, Product Description, Review Text, Helpfulness Score, and User Rating.

Once the collection of Reviews has been parsed, the pre-processing phase can begin. Each review is separated into a collection of Chunks by the Chunker class, which are defined by sentences and/ or noun-phrases. Each Chunk object hold enough information to identify it with a Review as well as information that will be collected later in the pre-processing and processing stages. At this point, each chunk
is then run through the part of speech tagger and polarity analyzers that save the proper information within the chunks.

We now have multiple data structures containing the parsed and chunked reviews, each with three binary polarity scores and both tagged and untagged text. The data is now ready for the third stage, the processing stage. Here, we begin to build our vector representation of each chunk. The vectors contain the features that we will use to classify the chunks via the Ridge Regression model. The features are explained in detail in section 3.2.1.

During the processing stage, a specific product and its user reviews will be processed and the Sentiment and Statistical Engines (Sentiment Analyzer and Statistics classes) calculate scores for each chunk and add them to the vector. The Sentiment Analyzer performs simple sentiment analysis, using sentiment values for individual words based on part of speech. The sentiment values are calculated by summing the maximum positive and negative sentiment values for the given word and its associated part of speech tag. The statistics class also adds word frequency, sentence length, and tf.idf.

Once the vectors have been calculated, we begin to build our positive and negative lists. For each product, a list of positive and negative chunks is generated by using the Ridge Regression model to classify each chunk’s vector. Since our data was limited as explained above, we returned a sorted list of all positive and all negative chunks as displayed below in the Example Section (Section: 4.3). Normally, we would add a second layer of clustering so that within each of the positive and negative lists we return as many unique aspects of the product as possible. We use similarity algorithms used in multi-document summarization to accomplish this task.

### 3.2.1 SentiSummary Features

Because our model uses a vector-based classification algorithm, the features used are very important. SentiSummary tries to use as many features as possible from both traditional statistical multi-document summarization and sentiment analysis. Below is a list of the 29 features used in SentiSummary:

- Unigram, Bigram, and Trigram Polarity Classification
- Amazon Helpfulness Rating
- Sentence Length
- Average and Log-Normalized Tf-idf
- Average and Log-Normalized Max Tf-idf
- Average and Sum Log Probabilities
- Top 5 Tf-idf and Log-Normalized Tf-idf
- Sum and Average of Each Noun, Verb, Adj, and Adv Sentiment Values (SentiWordNet scores)

The first and last bullets are sentiment features, while three through seven are statistical features. The sentiment features attempt to give insight into the opinions the reviewers’ expressed. The statistical features attempt to characterize the language, wording and semantics of the sentences themselves.

---

4tf.idf (term frequency-inverse document frequency) is a statistic that measures how important a word is to a document in a collection or corpus.

---

4Figure 1: GUI Initializer and Program GUI
3.2.2 Visual Output

Our visual output, which can be seen above in Figure 1, allows a user to easily generate and view summaries in an simple and intuitive graphical interface rather than use a cumbersome command prompt. The program starts up and initializes various models, including Lillian Lee's Polarity Model, the Stanford POS tagger, and the SentiWordNet file. Once complete, the main window loads which includes the following graphical elements: On the top there is a drop-down box with the list of categories and a search bar and button to search for products. Right below those elements there are two scroll panels. The one on left lists all of the products returned for a given search query and the one on the right displays the overall summary for a given product. The user first selects a category, enters some search terms in the search bar (or leaves it blank to list all of the products in the category), and then clicks the search button. The product list is then generated, and the user can select a product with the summary output displayed on the right.

The program may take a minute to initialize and train the models for the pre-processing and processing elements, as described above. The first search query may also take another minute, as the program parses all of the product chunks for a given category and caches them in a local data structure. However, all subsequent searches and summaries are virtually instantaneous.

3.2.3 Packages and Pre-built Systems

Because much of our research is based on the combination of existing technologies, we will be using some pre-built packages and tools to create our product review summarizer. The following is a description of each of those tools as well as any other specific details about the specific modules used. All of these are packages that we incorporated into our program. Thus it was necessary to build wrapper classes that instantiated the heart of the algorithms included in the packages.

- **Stanford POS Tagger:**
  The Stanford Log-Linear Part-of-Speech Tagger is a Java-implemented POS tagger that was developed by the Stanford Natural Language Processing Group. We used a pre-trained maximum entropy tagger that outputs tags based on the Penn Treebank tag set. The default pre-trained model used is a bidirectional model trained on the Wall Street Journal. When tested on other Wall Street Journal text, the performance of the model was over 97%, while only 89.3% on unknown words, as reported in the README-models.txt provided in the package [3].

- **SentiWordNet:**
  SentiWordNet is an opinion mining extension of WordNet. Each WordNet synset (with appropriate part-of-speech tags) is given a positive, negative and object sentiment score. For more information, please see the original paper on SentiWordNet entitled “SentiWordNet: A Publicly Available Lexical Resource for Opinion Mining” [2].

- **LingPipe:**
  LingPipe is the natural language processing API behind Breck Baldwin’s ThreatTracker. ThreatTracker was developed with funding from DARPA under a research grant for the Translingual Information Detection, Extraction, and Summarization (TIDES) program. We use LingPipe’s polarity analysis features and eventually the spell checking features. The Polarity Analyzer is based on the work of Pang and Lee [13] and uses their polarity data based on 1000 positive and 1000 negative full test movie reviews drawn from the Internet Movie Database’s archives. LingPipe provides a command line tutorial that was adapted to a Java class. The actually classifier uses the provided data with an n-gram language model classifier. As specified on the Sentiment Analysis tutorial posted on the LingPipe website, the classifier has approximately 81.5% accurate in its positive/negative classification.

3.3 System Performance

As a basis for understanding the current runtime of the product review summarizer, we ran our full system on all the available review data for electronics, which consisted of 22,979 reviews for 3,975 different products. We ran and averaged 5 full runs, from file-formatting to outputting all summaries to a file. We have divided the average total time per run into two different categories: pre-loading and per Product calculations. Pre-loading consists of runtime that was dedicated to loading of model files that were required before reviews were examined. These models included loading the POS tagger model, the Polarity models, Ridge Regression Model and SentiWordNet model. The review calculation runtime is composed of file formatting, parsing, pre-processing, summarizing and file outputting and is shown as an average time per product. The following are the results:

<table>
<thead>
<tr>
<th>Time</th>
<th>Pre-Load</th>
<th>Summarizing</th>
</tr>
</thead>
<tbody>
<tr>
<td>in Milliseconds</td>
<td>38.284</td>
<td>111.481</td>
</tr>
<tr>
<td>in Seconds</td>
<td>38.294</td>
<td>0.111</td>
</tr>
</tbody>
</table>

Table 1: Performance Table

The results show that when our program is loaded, it would need approximately 30 seconds to load all the necessary models to run the summarizer. Once the program is running and all models pre-loaded, it would take on average no more than one second to summarize and output a review. What this test is missing is that when we run the program with the GUI, we will search through all the reviews (or retrieve them from the Amazon API) and then process them. Thus, we would need to add some amount of time, upwards of one minute, on top of the 1 second calculation time.

An important note on the performance runs. They were run in Eclipse Galileo with JVM 1.6 and a 2 GB heap space on a 2.8 GHz Intel Core 2 Duo, 4 GB of RAM laptop running Mac OS X 10.6.3.

4. RESULTS

The following paragraphs describe the reasoning and results for our classifications, as well as some good and bad examples from our code.

---

1. All packages are either freely available on the web or under the GNU public license.
2. build: 20090920-1017
4.1 Classifiers

In order to classify our results accurately we manually tagged 1000 chunks as either positive or negative, skipping those chunks that we deemed neither positive or negative. We then used five different classifiers including Naive Bayes, Ridge Regression, Streamwise Regression, Stepwise Regression, and Perceptron, generating accuracy results for each of these classifiers. The following paragraphs explain more depth as to how we classified them, the accuracy results for each classification, and which classifier we ultimately chose.

In order to determine the polarity of a given chunk, a standard NLP technique is to break apart that chunk into a feature vector representation, use a classifier from pre-trained data, and determine the classification decision on this new chunk. We chose features that we thought would differentiate the chunks and help create a split distribution between positive and negative neutrality. A simple way to classify data is through linear regression. Linear regression allows one to predict the classification as well as possible as a linear combination of the features of a given chunk.

\[
\text{argmin}_w \sum_i \left( y_i - \sum_j w_j X_{ij} \right)^2
\]

Using a simple formula, we can determine the classifier and the chunk’s classification using the following equations:

\[
w = (X^T X)^{-1} X^T y
\]
\[
\hat{y} = x \cdot w
\]

In this case, w is the classifier and \(\hat{y}\) is classification for the given chunk (with feature vector representation as x). While this predicts a value, rather than a class label, that value if greater than zero, would imply positive polarity and if less than zero, would imply negative polarity.

We modeled three of our classifiers based on Linear Regression. The first classifier is Ridge Regression. Ridge Regression helps deal with the issue of collinearity when trying to invert the nearly singular \(X^T X\) matrix, which is because many of the features are highly correlated. By adding \(\lambda \cdot I\) with some value of \(\lambda\), the matrix is replaced with a more numerically stable matrix.

The next two classifiers are Streamwise and Stepwise Regression. Like Ridge Regression, we added the \(\lambda \cdot I\) to create a more stable model, but Streamwise only adds features to the model which reduces the penalized error. This is the greediest possible search, as compared to just simple Ridge Regression. Stepwise, while not quite as greedy as Streamwise, considers all possible features for addition to the model and picks the best one. It stops when the penalized error ceases to decrease.

The next classifier we used was a Naive Bayes classifier. A Naive Bayes classifier is a simple probabilistic classifier which uses Bayesian statistics and assumes independence between features. I.e. even if certain features depend upon one another, the NB classifier considers each property to independently contribute to the probability of the chunk being either positive or negative. We trained our classifier on the 1000 manually tagged chunks that I described above.

\[
p(C|F_1, \ldots, F_n) = \frac{p(C)p(F_1, \ldots, F_n|C)}{p(F_1, \ldots, F_n)}
\]

\[
\text{classify}(f_1, \ldots, f_n) = \text{argmax}_c p(C = c) \Pi_{i=1}^n p(F_i = f_i|C = c)
\]

The last classifier we used was a Perceptron model. The Perceptron algorithm starts with a randomly initialized hyperplane the same size as our feature vector. We then incrementally modify the hyperplane such that the points that are misclassified move closer to the actual classification to the training set. The perceptron algorithm stops when all learning examples are correctly classified (or when the Perceptron algorithm stops improving). This algorithm dates back to the 1950s and is the motivation behind Neural Networks. The one problem with the Perceptron algorithm is that if the data is not linearly separable (as in the problem with Linear Regression) then the hyperplane will never converge, and will continue to bounce all over the place.

Voted Perceptron works just like a regular Perceptron except that you keep track of all of the intermediate models and when it comes time to classify something, all of the models vote on the answer, and the algorithm takes the majority, preventing the “bouncing around” issue. While we attempted to use Voted Perceptron in our classification, we found that it did not work as expected, and instead, tested the accuracy on the other five classifiers.

4.2 Accuracy of Classifiers

In order to test the accuracy of the classifiers, we created a simple Python script which takes as input the 1000 manually tagged feature vectors, splits the data into 80% training set and 20% testing set, training on 800 vectors and testing on the other 200, and computing an error score. We did ten random seeds (for a different 80-20 split each time), and computed an overall average error rate as the accuracy for the classifier. It should be noted that there are approximately 66% positives and 34% negatives. Table 2 below displays these results.

As we can see from the overall percent correct for the training data, the percentages for all of the algorithms show that they do not significantly overfit the training data. This is important because it tells us that the classifiers learned as much as possible from the training data about general

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Positive</th>
<th>Negative</th>
<th>Overall</th>
<th>Positive</th>
<th>Negative</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>0.498</td>
<td>0.513</td>
<td>0.506</td>
<td>0.488</td>
<td>0.495</td>
<td>0.518</td>
</tr>
<tr>
<td>Ridge Regression</td>
<td>0.8</td>
<td>0.507</td>
<td>0.691</td>
<td>0.761</td>
<td>0.485</td>
<td>0.664</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>0.658</td>
<td>0.301</td>
<td>0.525</td>
<td>0.673</td>
<td>0.327</td>
<td>0.554</td>
</tr>
<tr>
<td>Streamwise Regression</td>
<td>0.875</td>
<td>0.271</td>
<td>0.649</td>
<td>0.876</td>
<td>0.256</td>
<td>0.658</td>
</tr>
<tr>
<td>Stepwise Regression</td>
<td>0.954</td>
<td>0.1</td>
<td>0.636</td>
<td>0.949</td>
<td>0.089</td>
<td>0.643</td>
</tr>
<tr>
<td>Perceptron</td>
<td>0.7</td>
<td>0.35</td>
<td>0.57</td>
<td>0.693</td>
<td>0.344</td>
<td>0.569</td>
</tr>
</tbody>
</table>

Table 2: Classifier Performance Table for both Training and Testing
rules without being overly specific. When compared to our baseline of a coin flip (“heads” it is positive and “tails” it is negative), all of the classifiers did significantly better on the positives, while the opposite is true for negative training vectors. As stated before, we used ridge regression, which is the median value on positives (80%) and statistically the same as the best on negatives (baseline and ridge regression).

Considering the test data, we see again that the baseline is lowest on the positives and except for ridge regression, baseline is highest on the negatives. This dichotomy is probably due to the 2-to-1 ratio of the training data. If we look more closely at the individual classifiers, we notice that NB and Perceptron algorithms do not work as well for the reasons described above. Since the data is not exactly separable, assuming independence does not necessarily return or improve the accuracy results. However, the other 3 regression algorithms are all pretty similar. It is interesting to note that simple Ridge Regression and Streamwise Regression produce the highest overall Train and Test success rates, while Stepwise produces lowest overall Train and Test success rates. This may be because having more features is actually beneficial in certain cases and/or splits of the data.

While Stepwise may not have returned the best overall average Train/Test success rate, the algorithm showed us which feature or features were most relevant in determining classification. For eight out of ten of the seeds, the only feature that Stepwise included was sentence length. In all of those cases, the size of the sentence length correlated to positive polarity. Thus, the more words in the chunk, the more likely the chunk is positive. The remaining Stepwise seeds included Lillian Lee’s Trigram and Unigram Polarity classifiers, the average adjective sentiment, and the 5th most important being tf.idf log normalized value. While the polarity classifiers make sense for important features, based on these results, it does appear that the SentiWordNet scores, in particular the adjective score, does help classify positive and negative chunks.

Although the overall correct rates are interesting, the rates for positive and negative separately give more insight into the best algorithms. Assuming an average positive success rate, it seems that the better the negative success rate, the better the overall rate. Optimally what would be best would be high rates for both positive and negative successes. Considering these results, a classification voting system where Ridge Regression, Streamwise Regression and Stepwise Regression voted might yield a success rate over 70%, which would be a significant improvement. All-in-all, the results give hope that with more refined features and other improvements described below in the Future Work section (Section 4.4), SentiSummary could produce very useful results.

Please see the appendix for the full output of the Python script (Appendix: B).

4.3 Examples

The following paragraph describes some examples that have been pulled from our output. Currently, our code is run on only electronics reviews. As we read through multiple reviews in all of the standard Amazon categories, we found that while some were objective and useful in parsing and classifying, others were just summaries of the products themselves (such as DVDs). Moreover, we chose to use only the electronics category as we felt the language in electronics reviews was less domain specific than other categories, such as movies and book. For instance, if a book review were to contain the phrase “read the book” that would imply a positive review. However, in a movie review, that same sentence would imply a negative review.

When we first classified the chunks as positive and negative, they were not that accurate since we did not use a classifier. Rather, we solely used SentiWordNet and the Unigram Model of Lillian Lee’s Polarity Classifier to determine the polarity of the chunk. Moreover, we did not calculate accuracy scores either. However, our current model is a lot more accurate and effective, and is a useful baseline for any future work.

4.3.1 Good Example

See above in Figure 2 to see a good example.

The positive scores are pretty accurate. The top three all show exactly what one would expect. What’s interesting is that the lowest positive score, the chunk, “I just got my Minidisc player in the mail today” has a score near zero, and is a pretty neutral chunk. Future work includes creating a neutrality classification to determine whether chunks are either Neutral OR Positive/Negative. The negative scores for this product are also somewhat accurate. The most negative one is exactly what would one expect, while the other negative score is more neutral. Still, this example is significantly more accurate than our previous model.

Figure 2: A Good Example: Sony Minidisc Network Walkman

Sony MZ-DX430PK/WH Psyc Minidisc Network Walkman (White): Electronics
Total Number of Reviews: 1

Prettifies:
Score: 69.66
It play great
Score: 63.96
Working great
Score: 49.61
Very simple and easy, just great tunes
Score: 18
I just got my Minidisc player in the mail today

Negatives:
Score: -42.79
Only complaint might be that there is no way to hook it up to power via AC adapter, but when you only takes one
Score: -30.50
I haven’t messed around too much with the software yet, but I put a 40 minute cd onto a mini disc at LP2 in about five minutes
4.3.2 Bad Example

See above in Figure 3 to see a bad example.

In this example, none of the positive scores should be classified as positive. The most positive is either negative or neutral, and we assume that it was labeled as positive given the word "favorable" and the length of the sentence. Moreover, the other three scores are also labeled as positive, most likely due to their length. The negative classifications, on the other hand, are pretty accurate. The one downside is that almost all of the data that we were given initially only had one review per product, thus we did not develop any code to weed out similar positives/negatives and choose the best ones.

4.4 Future Work

The first would be to create a more robust sentiment engine. Our current engine uses SentiWordNet sentiment scores for individual words, and takes the absolute value of both the positive and negative sentiment values for each word. One downside is that these entries are manual rather than automatic, and if multiple individuals entered differing sentiment values, SentiWordNet may not be accurate. Moreover, to calculate the sentiment for our chunks, we added up the absolute value of the positive and negative sentiment, so words with either polarity may skew the results. If there were some way to know if these words were used in a positive way or negative way that would improve our results.

The second manner would be including some sort of negation detection. For chunks that use negation words such as "not", the polarity for the sentence may or may not be reversed. Creating an automatic way to determine negation would greatly improve the accuracy of the overall chunk scores.

While Ridge Regression gives moderate results for classification, using a better classification mechanism such as Support Vector Machines would return more accurate results.

As described previously, we only included classifications for positivity vs. negativity. When we manually classified the 1000 sentences as positive or negative, we ignore many which either were fragments or were neutral. Our classifier may have been more effective if instead of skipping those sentences, we kept track of the neutral ones and created a classifier to determine neutral chunks vs. positive/negative chunks.

Finally, by including more lexical features for each chunk and increasing the number of manually tagged chunks from 1000 to 2000, our classifier would most likely have higher accuracy rates.

5. CONCLUSION

SentiSummary currently outputs a good baseline for further research in the area of sentiment analysis and summarization for user product reviews. It is our hope that we can continue to improve our methodology and SentiSummary as well as continue to incorporate the breakthroughs by other researchers in the areas of multi-document summarization and sentiment analysis. As we move further into the “Information Age” (or “Internet Age”), it is clear that data will always be in abundance. We as humans must rely on computer to help us sift through that information and gather the most important aspects and opinions. SentiSummary represents a first attempt to do just that.

6. REFERENCES


APPENDIX

A. EXAMPLES

The following examples are summarized by our product review summarizer from Dredze’s Amazon data. These examples are from only electronics products.

A.1 Good Example - Nintendo e-Reader

PRODUCT ID: B00006LELP

*Nintendo e-Reader - Game console trading card reader*

**Positives:**
- Score: 0.5 The only problem is to connect to games you need 2 Game Boys and a link cable but that it isn’t too much of a problem
- Score: 0.375 The e-Reader is one of the best things that happened to Nintendos GBA
- Score: 0.25 The only flaw is you need to scan 10 cards to play one NES game and you can only save one at a time
- Score: 0.125 Cards read once and never again

**Negatives:**
- Score: -0.5 We have been unsuccessful with this e-reader attachment since we purchased it
- Score: -0.25 The games are actually in the dot codes
- Score: -0.125 But the e-Reader does more than NES games

A.2 Bad Example - Zune 30 GB Digital Media Player

PRODUCT ID: B000H0QDCC

*Zune 30 GB Digital Media Player (Brown)*

**Positives:**
- Score: 0.75 The Zune is so much better
- Score: 0.5 won’t recognize the unit
- Score: 0.375 It has such a better feel to it and the controls react nicely, instead of making you frustrated like the iPod
- Score: 0.25 It is my way of using my music anywhere I go
- Score: 0.125 I went to the store to get some info on the Ipod, and luckily the departments store I frequent didn’t carry the unit, but offered the Zune

**Negatives:**
- Score: -0.625 Overall, the software and device are simple to use, quick, and deliver exactly the content I want without a hassle
- Score: -0.5 I have two iPods that don’t work
- Score: -0.25 I would check out the other models of media players, especially the Archos brand, for they have a bigger screen and hard drive
- Score: -0.16666666666666666 The Zune is far more superior than any Ipod model that I have either read about or experienced for myself
- Score: -0.125 I was more then dismayed
A.3  Sentence Issue Example - Lithium Watch Battery For CR2032
PRODUCT ID: B0007P9H48
Lithium Watch Battery For CR2032

Positives:
Score: -0.75 All for less than my local hardware store charges

Negatives:

A.4  Chunking Example - Samsung 226BW 22-Inch Digital/Analog Widescreen LCD Monitor
PRODUCT ID: B000NBWNU
Samsung 226BW 22-Inch Digital/Analog Widescreen LCD Monitor (Black)

Positives:
Score: 0.625 I replaced my 19"CRT with this monitor and love it. The color is sharp and games look great. The only thing is the stand does not tilt or adj up just pan so I bought a Ergotron desk mount. Samsung should have sent a better stand
Score: 0.25 Better check the policy from whom you buy from, but I hear Amazon is pretty lenient.

Negatives:
Score: -0.5 Samsung only takes the monitor back if you have SEVENTEEN dead pixels
Score: -0.375 Bottom line: The Samsung 226BW is definitely worth the purchase and I know you won’t be disappointed

B. PYTHON SCRIPT OUTPUT

Initializing new Dataset...
Initialized dataset:
hyperplane_neg_train_file.model vs. hyperplane_pos_train_file.model
1000 examples

************
* BASELINE *
************
Seed 1:
Generating random 80-20 split...
Baseline Train Error Rate: 0.478097622028
Baseline Test Error Rate: 0.49
Seed 6:
Generating random 80-20 split...
Baseline Train Error Rate: 0.49436795995
Baseline Test Error Rate: 0.485
Seed 7:
Generating random 80-20 split...
Baseline Train Error Rate: 0.501877346683
Baseline Test Error Rate: 0.46
Seed 8:
Generating random 80-20 split...
Baseline Train Error Rate: 0.489361702128
Baseline Test Error Rate: 0.475
Seed 9:
Generating random 80-20 split...
Baseline Train Error Rate: 0.514392991239
Baseline Test Error Rate: 0.525
Seed 10:
Generating random 80-20 split...
Baseline Train Error Rate: 0.468085106383
Baseline Test Error Rate: 0.43

************
* NAIVES BAYES *
************
Seed 1:
Generating random 80-20 split...
Train Error Rate: 0.485607008761
Test Error Rate: 0.43
Seed 2:
Generating random 80-20 split...
Train Error Rate: 0.480600750939
Test Error Rate: 0.455
Seed 3:
Generating random 80-20 split...
Train Error Rate: 0.479349186483
Test Error Rate: 0.425
Seed 4:
Generating random 80-20 split...
Train Error Rate: 0.473091364205
Test Error Rate: 0.465
Seed 5:
Generating random 80-20 split...
Train Error Rate: 0.469336670839
Test Error Rate: 0.44
Seed 6:
Generating random 80-20 split...
Train Error Rate: 0.473091364205
Test Error Rate: 0.43
Seed 7:
Generating random 80-20 split...
Train Error Rate: 0.4857008761
Test Error Rate: 0.425
Seed 8:
Generating random 80-20 split...
Train Error Rate: 0.479349186483
Test Error Rate: 0.425
Seed 9:
Generating random 80-20 split...
Train Error Rate: 0.469336670839
Test Error Rate: 0.455
Seed 10:
  Generating random 80-20 split...
  Train Error Rate: 0.475594493116
  Test Error Rate: 0.44

******************************
* RIDGE REGRESSION *
******************************
Seed 1:
  Generating random 80-20 split...
  Train Error Rate: 0.317897371715
  Test Error Rate: 0.295
Seed 2:
  Generating random 80-20 split...
  Train Error Rate: 0.30788485607
  Test Error Rate: 0.345
Seed 3:
  Generating random 80-20 split...
  Train Error Rate: 0.312891113892
  Test Error Rate: 0.34
Seed 4:
  Generating random 80-20 split...
  Train Error Rate: 0.304130162703
  Test Error Rate: 0.35
Seed 5:
  Generating random 80-20 split...
  Train Error Rate: 0.305381727159
  Test Error Rate: 0.345
Seed 6:
  Generating random 80-20 split...
  Train Error Rate: 0.312891113892
  Test Error Rate: 0.34
Seed 7:
  Generating random 80-20 split...
  Train Error Rate: 0.294117647059
  Test Error Rate: 0.365
Seed 8:
  Generating random 80-20 split...
  Train Error Rate: 0.315394242804
  Test Error Rate: 0.35
Seed 9:
  Generating random 80-20 split...
  Train Error Rate: 0.305381727159
  Test Error Rate: 0.36
Seed 10:
  Generating random 80-20 split...
  Train Error Rate: 0.315394242804
  Test Error Rate: 0.29

******************************
* STREAMWISE REGRESSION *
******************************
Seed 1:
  Generating random 80-20 split...
  sentence length 0.371714643304
  Breaking...
  Train Error Rate: 0.371714643304
  Test Error Rate: 0.365
Seed 2:
  Generating random 80-20 split...
  sentence length 0.377972465582
  Breaking...
  Train Error Rate: 0.377972465582
  Test Error Rate: 0.33
Seed 3:
  Generating random 80-20 split...
  isPositive Trigram 0.382978723404
  avg adj sent 0.339173967459
  isPositive Unigram 0.332916145181
  #5 tfidf log norm 0.329161451815
  Breaking...
  Train Error Rate: 0.329161451815
  Test Error Rate: 0.38
Seed 4:
  Generating random 80-20 split...
  sentence length 0.376720901126
  Breaking...
  Train Error Rate: 0.376720901126
  Test Error Rate: 0.33
Seed 5:
  Generating random 80-20 split...
  sentence length 0.366708385482
Breaking...
Train Error Rate: 0.366708385482
Test Error Rate: 0.37
Seed 6:
Generating random 80-20 split... sentence length 0.369211514393
Breaking...
Train Error Rate: 0.369211514393
Test Error Rate: 0.365
Seed 7:
Generating random 80-20 split... sentence length 0.369211514393
Breaking...
Train Error Rate: 0.369211514393
Test Error Rate: 0.365
Seed 8:
Generating random 80-20 split... sentence length 0.365456821026
Breaking...
Train Error Rate: 0.365456821026
Test Error Rate: 0.375
Seed 9:
Generating random 80-20 split... sentence length 0.366708385482
Breaking...
Train Error Rate: 0.366708385482
Test Error Rate: 0.375
Seed 10:
Generating random 80-20 split... isPositive Trigram 0.376720901126
avg adj sent 0.347934918648
sum verb sent 0.344180225282
Breaking...
Train Error Rate: 0.344180225282
Test Error Rate: 0.33

****************
* PERCEPTRON *
****************
Seed 1:
Generating random 80-20 split...
Train Error Rate: 0.371714643304
Test Error Rate: 0.365
Seed 2:
Generating random 80-20 split...
Train Error Rate: 0.46307848561
Test Error Rate: 0.425
Seed 3:
Generating random 80-20 split...
Train Error Rate: 0.423028785982
Test Error Rate: 0.44
Seed 4:
Generating random 80-20 split...
Train Error Rate: 0.377972465582
Test Error Rate: 0.35
Seed 5:
Generating random 80-20 split...
Train Error Rate: 0.392991239049
Test Error Rate: 0.425
Seed 6:
Generating random 80-20 split...
Train Error Rate: 0.394242803504
Test Error Rate: 0.435
Seed 7:
Generating random 80-20 split...
Train Error Rate: 0.396745932416
Test Error Rate: 0.39
Seed 8:
Generating random 80-20 split...
Train Error Rate: 0.410513141427
Test Error Rate: 0.43
Seed 9:
Generating random 80-20 split...
Train Error Rate: 0.632040050063
Test Error Rate: 0.62
Seed 10:
Generating random 80-20 split...
Train Error Rate: 0.438047559449
Test Error Rate: 0.44

---------------------------------------------------------
Test | \% Correct Pos | \% Correct Neg | \% Overall
Base | 0.487965185019 | 0.494936965333 | 0.5175
Ridge | 0.760510803956 | 0.485168075709 | 0.6635
NB | 0.673492794358 | 0.327458043759 | 0.5535
Stream | 0.87615642388 | 0.255748210838 | 0.658
Step | 0.948648064028 | 0.0889803183496 | 0.6425
Perceptron | 0.692752282943 | 0.34392810131 | 0.569

Train | \% Correct Pos | \% Correct Neg | \% Overall
Base | 0.497678430666 | 0.513354391199 | 0.50550683605
Ridge | 0.800354434288 | 0.507070505512 | 0.690863579474
NB | 0.658447953978 | 0.301119704373 | 0.52478097622
Stream | 0.874655073252 | 0.271052887811 | 0.649311639649
Step | 0.954051927616 | 0.10037620001 | 0.636295369212
Perceptron | 0.700367646234 | 0.350675410238 | 0.569962453066