Spring 2013

Predicting Product Review Helpfulness Using Machine Learning and Specialized Classification Models

Scott Bolter
San Jose State University

Predicting Product Review Helpfulness Using Machine Learning and Specialized Classification Models

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment of the Requirements for the Degree

Master of Science

by

Scott Bolter

May 2013

The Designated Project Committee Approves the Project Titled

PREDICTING PRODUCT REVIEW HELPFULNESS USING MACHINE LEARNING AND SPECIALIZED CLASSIFICATION MODELS

by

Scott Bolter

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2013

Dr. Robert Chun Department of Computer Science

Professor Ronald Mak Department of Computer Science

Dr. Teng Moh Department of Computer Science

ABSTRACT

PREDICTING PRODUCT REVIEW HELPFULNESS USING MACHINE LEARNING AND SPECIALIZED CLASSIFICATION MODELS

by Scott Bolter

In this paper we focus on automatically classifying product reviews as either helpful or unhelpful using machine learning techniques, namely, SVM classifiers. Using LIBSVM and a set of Amazon product reviews from 25 product categories, we train models for each category to determine if a review will be helpful or unhelpful. Previous work has focused on training one classifier for all reviews in the data set, but we hypothesize that a distinct model for each of the 25 product types available in the review dataset will improve the accuracy of classification.

Furthermore, we develop a framework to inform authors on the fly if their review is predicted to be of great use (helpful) to other readers, with the assumption that authors are more likely to rethink their review post and amend it to be of maximum utility to other readers when given some feedback on whether or not it will be found helpful or unhelpful.

Using past research as a baseline, we find that specialized SVM classifiers outperform higher level models of review helpfulness prediction.

ACKNOWLEDGEMENTS

Thanks are due to Dr. Moh of the Department of Computer Science for his excellent guidance and dedication to the success of his students in the master's program.

Additionally, this research would not have been possible without the efforts of Mark Dredze and John Blitzer, who made the Amazon Product Review dataset available for download. This is appreciated a great deal.

Lastly, thanks are due to committee members Dr. Robert Chun and Ronald Mak for their time, guidance, and devotion to the students at San Jose State University's Department of Computer Science.

Table of Contents

List of Algorithms, Figures, and Tables

NOTE: All code and pseudo-code appearing in algorithm sections as well as in explanations appear in `courier type font`

Introduction

An increasingly prevalent trend in the sale of goods is the shift to e-commerce, or online shopping. Numerous, if not most, traditional “brick and mortar” stores have online shops where consumers can place orders of many of the same products they would find at the physical store locations. As this trend continues (often to the disdain of in-store workers), these locations have become simple “showrooms” where customers can see and touch the product, but actually plan to order it online where it may be cheaper, more varied in size or color, or simply more convenient to have shipped rather than brought home. Aside from convenience and competition, the largest benefit to customers is arguably the availability of firsthand reviews and feedback from other shoppers. “What do the reviews say?” and “How many stars did it get?” are questions that online consumers factor in to their purchasing decisions. In addition to the customer benefit, companies making the products being sold also benefit from online availability of such reviews. They can incorporate the feedback of their customers into future product iterations with the end goal of increasing sales. For these reasons, it is of high importance to strive for the best quality and most accurate reviews. One way to judge quality and accurate reviews is by their *helpfulness* to other readers, which is where this project focuses.

Currently, one of the most popular multi-category online shop is Amazon.com. With many products in many different departments, it has become a hugely popular option for online shoppers. This popularity increases the number of customer reviews which in turn adds to the site’s utility. Aside from a “star rating” from 1 to 5, customers can also submit textual feedback and product accounts, made available on the

product's page on Amazon.com. Next to each review are three simple user-interface elements: a label, "Was this review helpful to you?", and two buttons, "Yes" and "No". It is this mechanism that allows users to vote up or down the helpfulness of a product review. The website then allows customers to sort reviews by their voted helpfulness (the site's default review ordering) or temporally. While providing an excellent option for customers to filter out "good" and "bad" reviews, the problem with this system is the necessity of participation from review readers and the possibility that reviews that were not voted on or were authored so long ago they are not high up in the ordering of reviews. This means that helpful reviews would likely not be seen by customers unless they enumerated through a potentially very large set of other reviews.

To mitigate the issues mentioned above, this project develops a machine learning technique to automatically classify product reviews as either helpful or unhelpful, without the need for voting. This is done using Support Vector Machine (SVM) classifiers, with two end goals: to automatically classify reviews using only the review text and to provide a framework for classifying input text on-the-fly that informs review authors that their review is likely to be seen as either helpful or unhelpful. Our hypothesis is that if users are made aware of their review's predicted utility (or lack thereof) at the time they author it, they are more likely to correct or augment their reviews which will increase their helpfulness.

SVM Classifiers

Support Vector Machine classification is a type of supervised machine learning technique. The difference between a supervised versus an unsupervised machine

learning technique lies in the data used to create the classifier (model). In a supervised learning environment, the classifier is taught using data that has already been determined to reside in a specific class. The model is created to then predict which class a datum resides in based on the attributes of that datum. On the other hand, for unsupervised learning, commonalities between the data's attributes are used to cluster them into like groups, with no predetermined classification [13].

Using selected features of the data, an SVM classifier attempts to create a hyperplane that accurately divides the data into distinct classes. While this hyperplane could be imagined in many dimensions, for visualization purposes, it is simplest to visualize this in two dimensions, demonstrated by Figure 1 below.

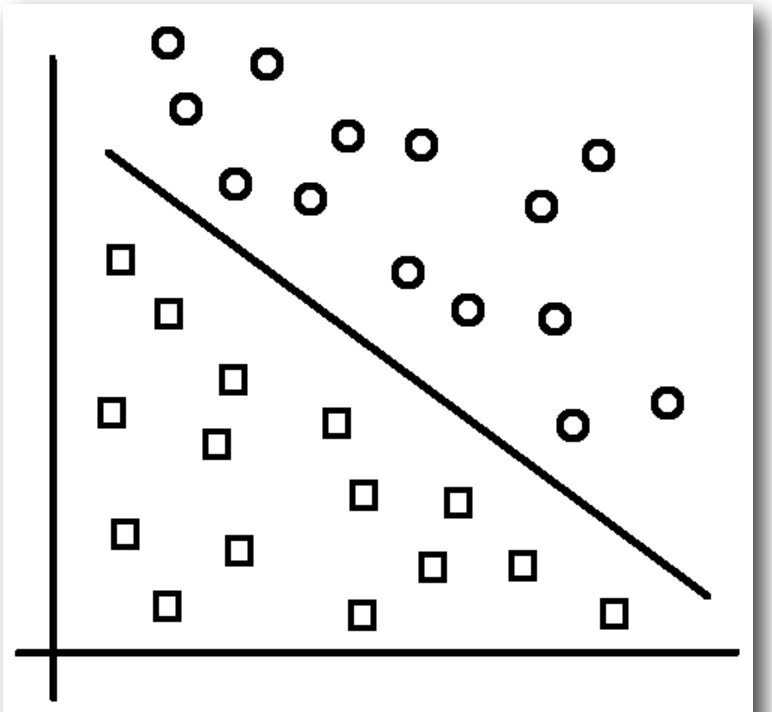


Figure 1: SVM Hyperplane in Two Dimensions

Here, the plane divides circles and squares into separate classes. When testing new data, any data point that is above the plane would be classified as a circle, with anything below the plane classified as a square [7].

While this is very simple to see in a cartesian plot of data, the same technique can be applied to problems with many dimensions. In the case of textual-based document classification, vectors are constructed using the document's inclusion of words in a global dictionary. The global dictionary is a collection of each unique word used in all of the training documents. We can then construct vector representations of each of the training documents using the indexes of the elements of the global dictionary and either a '1' for that index if the training document includes the word at that index, and a '0' otherwise. See Figure 2 below for a simple example.

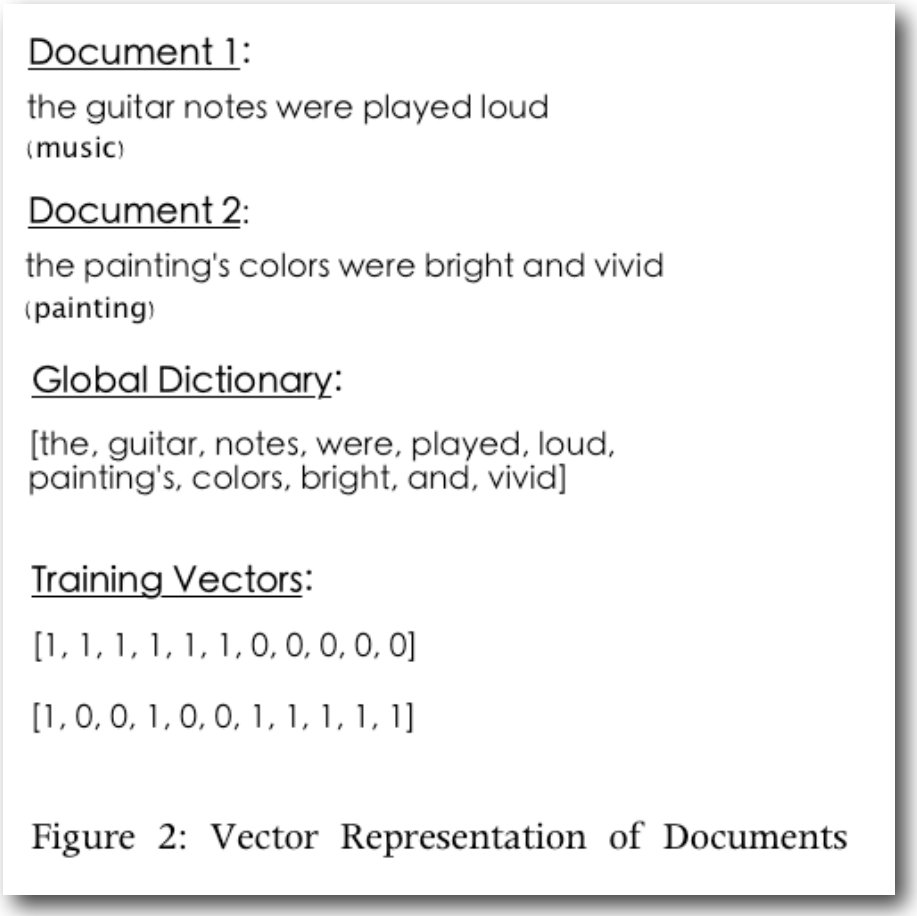


Figure 2: Vector Representation of Documents

Here we have two documents. Document 1 pertains to music, while Document 2 pertains to a painting. The global dictionary is an array of all words used in both documents. Each training vector is a representation of the document with a 1 placed in the index corresponding to the word's index in the global dictionary if present in the document, and 0 placed in the index of all words in the global dictionary that are not in the document. Note that the global dictionary contains only one 'the' since it is a collection of the *unique* words. Using this technique, each word in the global dictionary is considered a "feature" of the document. Comparing the features of every document against those used for training where the class is known informs the classifier about which class the document belongs to. In the figure above, for example, after training an SVM classifier, the document, "the notes were loud" would be predicted as music because the features of the document more closely match those of the training set (in this case, only Document 1) that were known to be pertaining to music [7].

Given the simple example above, one might notice that the two documents, known to be in different classes, contain common words, or, features. One can see that "the" and "were" are in both documents. These features do not add any benefit to our model because they offer no information gain since their presence in a test document could mean the document belongs to either class. For this reason, we ignore them. These common words have no weight at all. They are known as "stop words". To keep them from corrupting the model, the training routine is supplied with a set of stop words and each of them is filtered out from the training documents and global dictionary. The following figure shows what our example looks like after pruning common stop words.

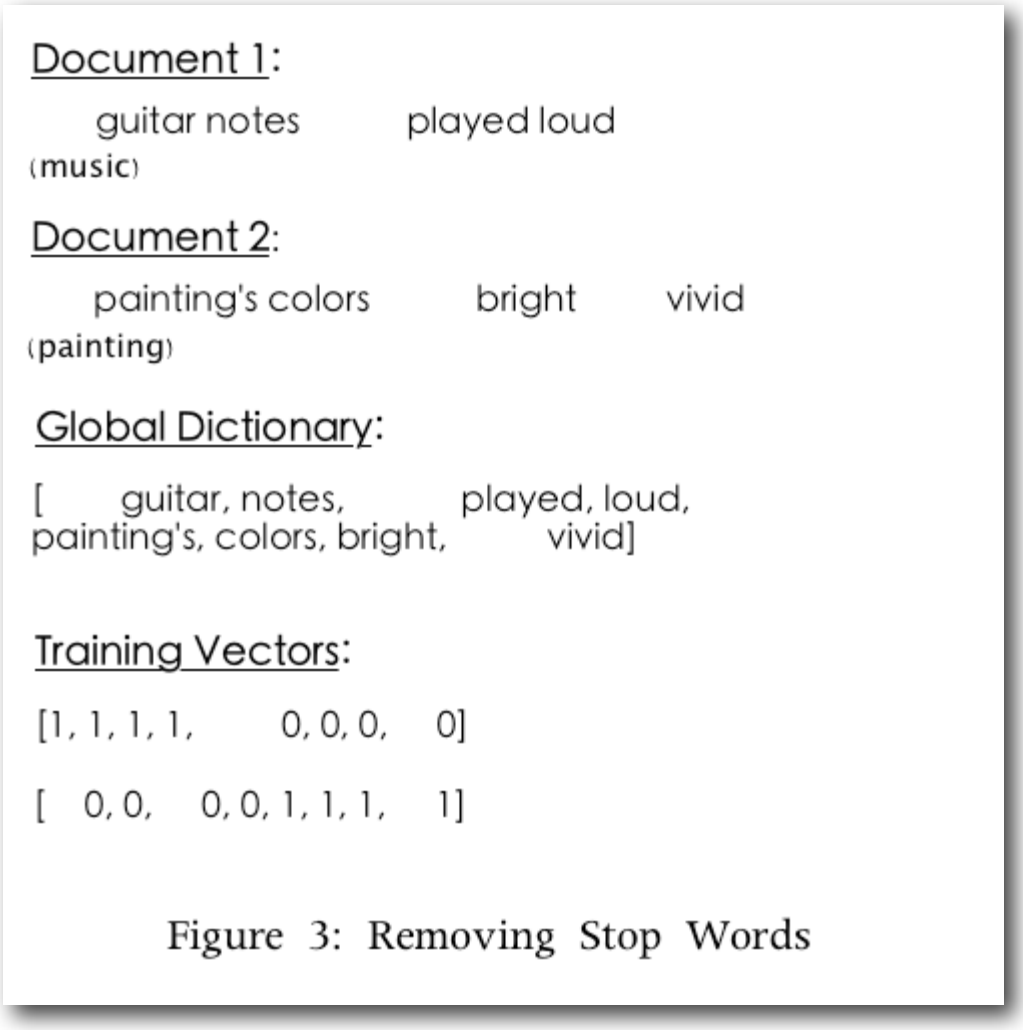


Figure 3: Removing Stop Words

Note that our global dictionary now contains words that have self-contained meanings more closely identifiable with the two classes as opposed to less meaningful words like “the”, “and”, and “were”. Also note that, while “and” was not common amongst both documents, it was still removed as it is common enough to warrant membership of our stop word list [1].

Lastly, and perhaps less of an obvious gain in our example, is word stemming. One might wonder about the different permutations of a word. For example, “played” is in our global dictionary, but what about “play”, “plays”, “playing”, “player”, “players”, etc.

Should not each of these words be associated with music rather than painting? To accomplish this, we employ another technique, known as “word-stemming” when training our SVM classifier. This is a simple matter of reducing every word to its root so that all permutations of the word will map to the same feature in the global dictionary. Figure 4 shows what our global dictionary will look like once every word is stemmed. The stemming of the documents’ features is omitted as they are subsets of the global dictionary.

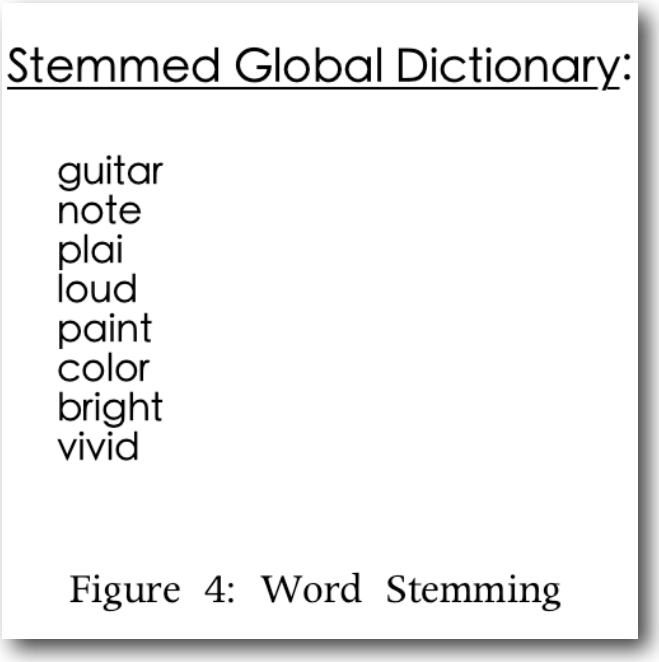


Figure 4: Word Stemming

As can be seen by “plai” in index 3 of the new global dictionary, the stemmed words are not necessarily defined words in the traditional sense of language. As long as the stemming of all permutations results in this same root, the model can be trained with this word as a feature.

SVM classifiers generally have a proven track record of success in document analysis and classification, especially in problems with a small number of target classes.

While there is empirical evidence in some research where some preprocessing has a negative effect on accuracy [2], when augmented with the two preprocessing techniques outlined above, stop word removal and word stemming, the accuracy of SVM classifiers typically improves, as we see in this project. In this research, the class of a document (review) is either helpful or unhelpful, and several studies indicate that SVM classifiers are effective in the problem space of product review classification.

Previous Work

As the prevalence of online shopping and product reviews have increased, so too has the interest in generating the best set of reviews to interested parties: customers, vendors, and producers of the goods being sold. Customers rely on this data to make purchasing decisions. Vendors recognize that the more robust their collection of product reviews, the more traffic will filter through their site, thus yielding a beneficial cycle of more customers leaving more product reviews. Finally, producers of the goods being sold can use these product reviews to inform design decisions and future product direction to ultimately appeal to more customers, essentially using the reviews to continue doing what customers favored, and alter areas where the products were reviewed poorly.

Blitzer et al [3] procured a dataset of Amazon.com product reviews from 25 different product categories. It has been used by several other studies and is used in this research. Their original research focused on detecting the sentiment, or general emotions, of the review to classify positive versus negative experiences with the product. This mirrors the aim of our research as they used features of the review other

than the actual product rating given by the reviewer to detect a positive (high) or negative (low) rating. Here, we use features other than the helpful or unhelpful votes that a review receives to ultimately determine if it would be found as helpful or not to other readers.

Kim et al [8] presented research in which SVM regression was employed to determine which features of the reviews from an Amazon.com dataset yielded themselves to helpfulness predictions. These features included the review length, unigrams (each word of the review text taken as a distinct feature, as we have done in this research), and the product ratings.

While the most commonly known form of spam is unwanted electronic mail, product review spam is also prevalent amongst online shopping review forums. This can take the form of unrelated links, advertising mixed into product review text, or false reviews, perhaps used to artificially increase the rating of a product. Lau et al [9] employed text mining and probabilistic language modeling to detect spam amongst review sets.

Liu et al [10] offered research on detecting low-quality reviews using different types of biases, suggesting methods to simply strip these reviews from the available set yielding a ground-truth opinion of the specific product being reviewed.

Hong et al [6] developed an Automatic Helpfulness Voting, coined “AHV”, system by building a ranking SVM classifier to assign a score to each of the reviews being tested and rank them in order of helpfulness. They built upon earlier successes with SVM classification by attempting to learn user preferences within their models. These include information needs fulfilled by product reviews, the credibility of reviews, and

each reviews' consistency with the mainstream opinion of the product. They find that when including these aspects, AHV performance is improved.

Here, we attempt to further the successes of SVM classification by using trained models for detecting helpful versus unhelpful reviews. By training multiple classifiers for the multiple product categories, we aim to improve accuracy predictions and believe that, intuitively, many models that have increased specificity should yield more accurate results. By developing an application to efficiently train, test, and store these classifiers, we gain the benefit of model multiplicity, effectively drilling down from a high-level model for a set "product reviews" to a more refined model for reviews of a specific product type such as "camera and photo".

Review Helpfulness Prediction With Specialized Models

Intuitive Idea

As mentioned above, SVM classification has shown to be a well-performing tool for classifying product reviews. As with this research, Hong et al and Kim et al, both presented SVM classification results on sets of Amazon.com product reviews. While their datasets certainly used reviews from various product types, the models trained were trained using reviews from the entire set. In this project, we hypothesize that accuracy could be improved if many models were trained amongst each product type available, thereby creating "specialized models". The intuitive idea lies in the inherent differences of a global dictionary between each product type. For example, the terms "size" and "fit" might be very important when determining the helpfulness of a review of

a pair of denim jeans, but are likely offer no information gain when dealing with reviews of something like a laptop. Similarly, including “RAM” and “CPU” in a global dictionary would only serve to pollute a classifier attempting to predict the utility of clothing reviews. With more refined global dictionaries and specific features of helpful or unhelpful reviews for a given product type, accuracy should increase because there are smaller instances of noise or corruption while training the models.

Pairing these specialized classifiers with persistent storage, we also present a framework for testing reviews prior to submission with the assumption that authors will be more likely to submit reviews that are more helpful if they are presented with the prediction of helpfulness at the time they submit them. Each of our 25 product types can store any number of classifiers. Each classifier has an accuracy associated with it. The classifier with the highest accuracy will be used to predict the helpfulness of a new review of a product in that product type. This is outlined further in later sections.

Dataset

Our dataset was obtained from the research done by Blitzer et al [3]. These authors made it available for download at http://www.cs.jhu.edu/~mdredze/datasets/sentiment/. The data is entitled “Multi-Domain Sentiment Dataset” and this research makes use of version 2.0. It is a collection of Amazon.com product reviews pulled from multiple product categories (domains). Some product types, such as books and DVDs, have hundreds of thousands of product reviews while others, such as musical instruments may have only a few hundred. Table 1 below outlines the dataset at the level of category and count.

Table 1 - Multi-Domain Sentiment Dataset Count

Product Category	Number of Reviews
apparel	9252
automotive	736
baby	4256
beauty	2884
books	975194
camera & photo	7408
cell phones & service	1023
computer & video games	2771
dvd	124438
electronics	23009
gourmet food	1575
grocery	2632
health & personal care	7225
jewelry & watches	1981
kitchen & housewares	19856
magazines	4191
music	174180
musical instruments	332
office products	431
outdoor living	1599
software	2390
sports & outdoors	5728
tools & hardware	112
toys & games	13147
video	36180

Each product category is represented in this dataset in an XML file containing all reviews for that category. Each review in the file is represented between *<review>* and *</review>* tags. All reviews have the attributes listed in Table 2 below.

Table 2 - Review Attributes

Attribute Name
product_name
product_type
helpful
rating
title
date
reviewer
review_text

One early decision in this project was to use a MySQL database to store the review information rather than flat files. This was done for several reasons. The first, and most practical, was for ease of organization and auditing. Opening even one product category's review set, even with a lightweight editor such as vi, proves very taxing on a computer. An early lesson was to *turn off XML syntax highlighting* when dealing with files of such magnitude. Simply loading all the data into memory when opening the XML file could take minutes for categories such as books and DVDs. Searching for a specific string was also very slow to complete. Aside from alleviating the frustrations of dealing with very large flat files, a MySQL database offers simplicity,

scalability, stable performance, and flexibility when considering programming options for library plugins in various languages.

However, for some of the challenges mentioned above, converting the XML files into a MySQL database proved non-trivial. Nokogiri (http://nokogiri.org/) was used to parse the XML files. This is a plugin, or "gem", for the Ruby programming language. It has excellent support for parsing both HTML and XML using XPATH and CSS3 Selector. While this library is useful for opening an XML document and parsing its contents, we ran into the same problem of memory constrains. Note that some of the product categories contain hundreds of thousands of reviews, each with the attributes listed above, yielding very large files. For example, the XML file containing all book reviews in this dataset is 1.3 gigabytes. Parsing this file in one shot proves taxing for even machines with respectable computing power. To mitigate this challenge, we employ a "divide and conquer" approach. Constructing a relatively simple shell script, *divider.sh*, we are able to specify the number of reviews we wish to have in a single file and the original XML file name as arguments and divide the product reviews for each category into several files, making them much more manageable for our parser. As outlined later, modifications to our file importer were made to allow specifying a directory which contains all of the files for a given product category rather than a single, much larger, XML file. From here, all reviews are able to be parsed and inserted into a MySQL table.

Application

Aside from using Ruby and Nokogiri to parse the review XML files into our MySQL database, the implementation of the rest of the application was done with Ruby. Specifically, Ruby on Rails, an object-oriented language with a framework for quickly setting up a skeleton of a web application, complete with drivers for connecting to MySQL. The application is named RHP, for Review Helpfulness Predictor. It handles everything from importing files or directories of files into the database, displaying each review and its attributes in a web browser, setting parameters for training a model, training and saving the SVM classifier, testing each classifier, and reloading classifiers from persistent storage to train a single user-entered review (one that is not in the database), providing on-the-fly feedback of a user's review's helpfulness.

Design

Architecture

As a framework, Ruby on Rails lends itself to development of a Model View Controller (MVC) application. This design pattern is appropriate for RHP for several reasons. A graphical component, able to display results of tests and statistics about our classifiers is desired simply for ease of use. However, there are different ways we wish to display the the models (from here on, in this section "models" will refer to the M of MVC, and classifiers will be used to refer to our machine learners to avoid confusion) of our application, whether they are reviews, product categories, or classifiers. The MVC design pattern is excellent at accomplishing this feat as we can reflect changes to our models across several views, in the case of Rails, web pages. Each view gives us a

different angle and context about the models, but no view has to be updated independently as a change in the model is reflected everywhere in the MVC architecture. Additionally, this architecture allowed the implementation to evolve over time as we discover new uses, attributes, or functions for the models. For example, throughout testing, there were instances where we wished views to display integer counts of how many reviews of a specific product type where voted as helpful, how many voted as unhelpful, and how many were not voted on at all by Amazon.com users. This can be calculated from the reviews table query based on the product type, but once it was clear this data was a common need, our product type model was updated with new attributes (columns in the MySQL table) such as *helpful_total* and *unhelpful_total*. After this change, these attributes only needed to be calculated once, and could then be accessed in constant time from several different views. Figures 5 and 6 below outline the MVC architecture in UML specific to two of the main models of the application.

Figure 5: Review MVC

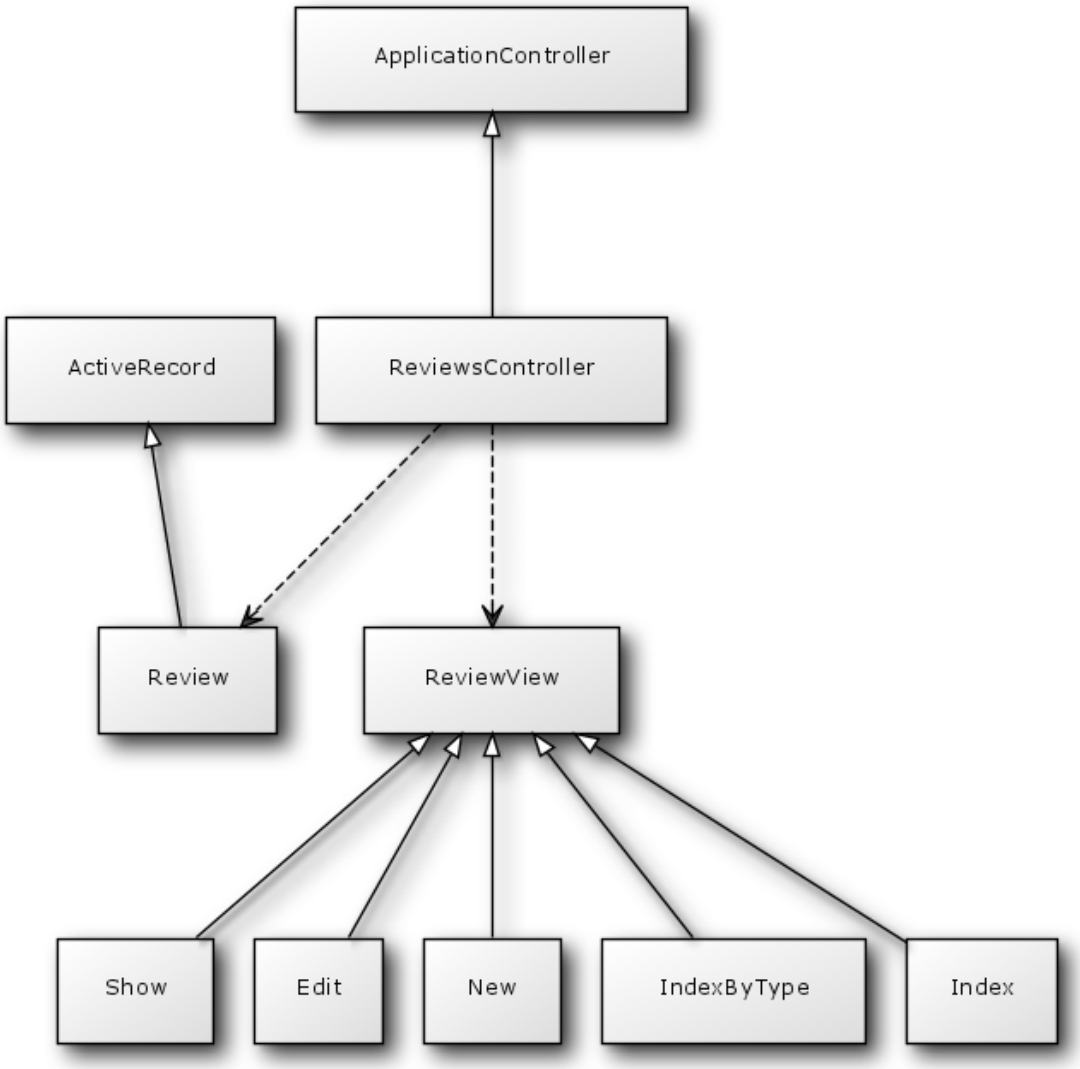
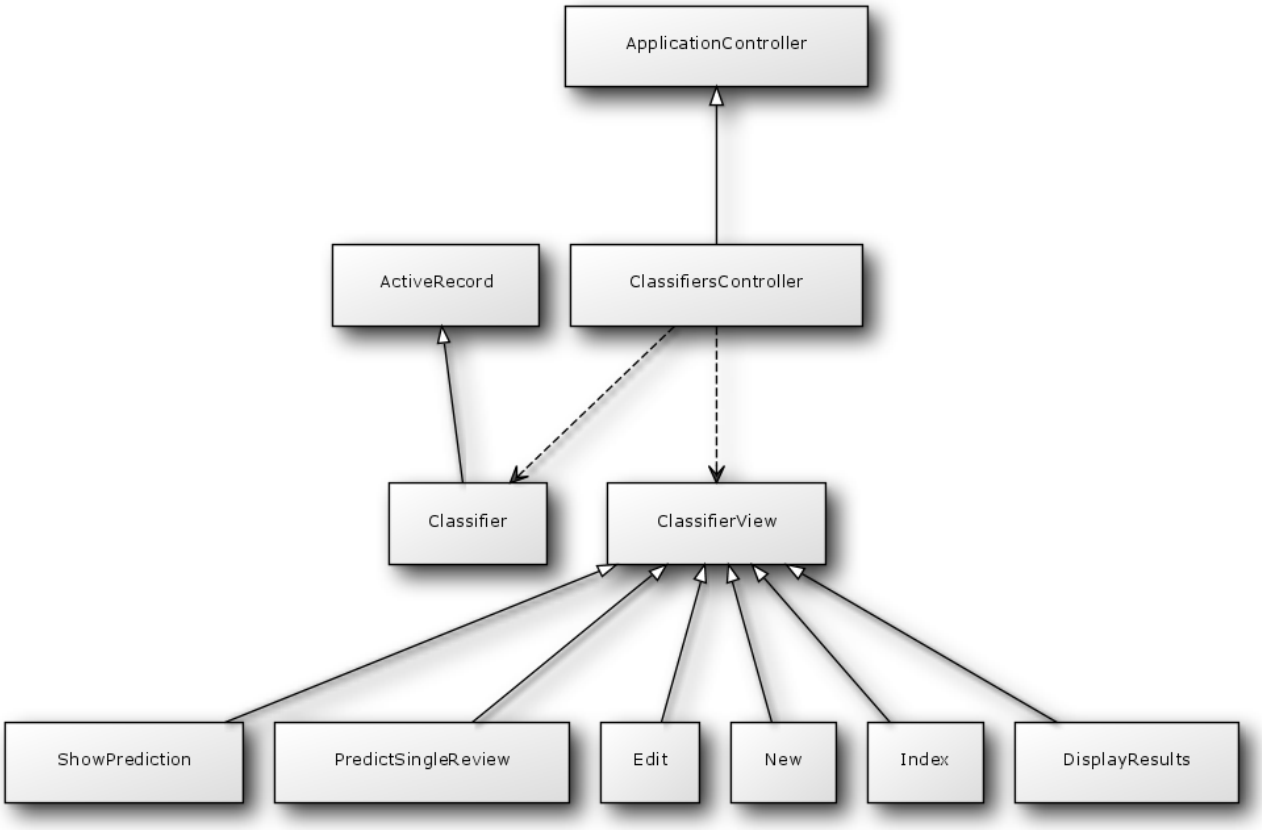


Figure 6: Classifier MVC



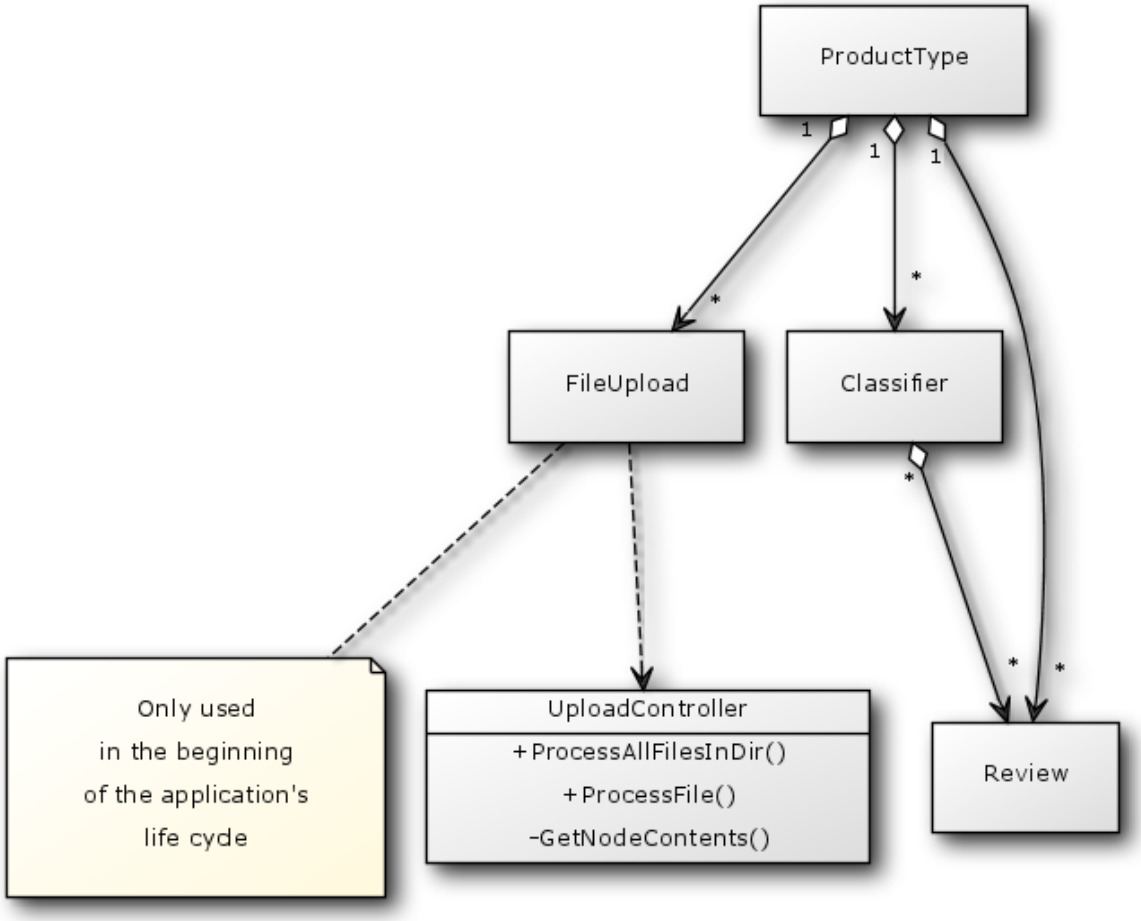
Both models above inherit from the ActiveRecord base class. In Ruby on Rails, this class abstracts interactions with a SQL database. For example, rather than using a SELECT statement with a LIMIT of 1 to obtain the first review from the reviews table of the MySQL database, one can simply call `Review.first` and the Review class will make use of ActiveRecord's methods to perform the SQL statement behind the scenes, simplifying functions requiring access to the database. Similarly, each of the models has a dedicated controller which inherits from ApplicationController. This parent class abstracts the methods necessary to handle RESTful calls, URLs forwarded by the web server to the controllers which contain GET and POST methods along with parameters. Parsing these parameters, forwarding to the appropriate method, and redirecting to the

requested views is all abstracted by this parent class, allowing both ReviewsController and ClassifiersController to remain succinct and clear classes in the application with added functionality specific to their respective models and views.

Classes

The application's models are constructed with two primary relationships, association and aggregation. While the product type model is not mentioned in the MVC architectures above because there are no views or a controller associated with it, it is still one of the three main models of the application. In Rails, these relationships are easily maintained using metadata in each of the model's class files. The metadata is a tag in the form of `has_many` (aggregation) and `belongs_to` (association). Each of these flags are followed by the name of the model to which this class either has many of or belongs to. These allow quick access to instances of the aggregating or associating classes from the aggregated or associated class. For instance, a review belongs to a product type and a product type has many reviews. So in our review class, stipulating at the top of the class file, `belongs_to :product_type`, allows all instances of the review class to quickly access the product type to which it belongs using the dot operator. For example, `some_review.product_type` will yield the instance of the product type class to which `some_review` belongs. The following diagram in Figure 7 illustrates these relationships in UML.

Figure 7: Model Relationships



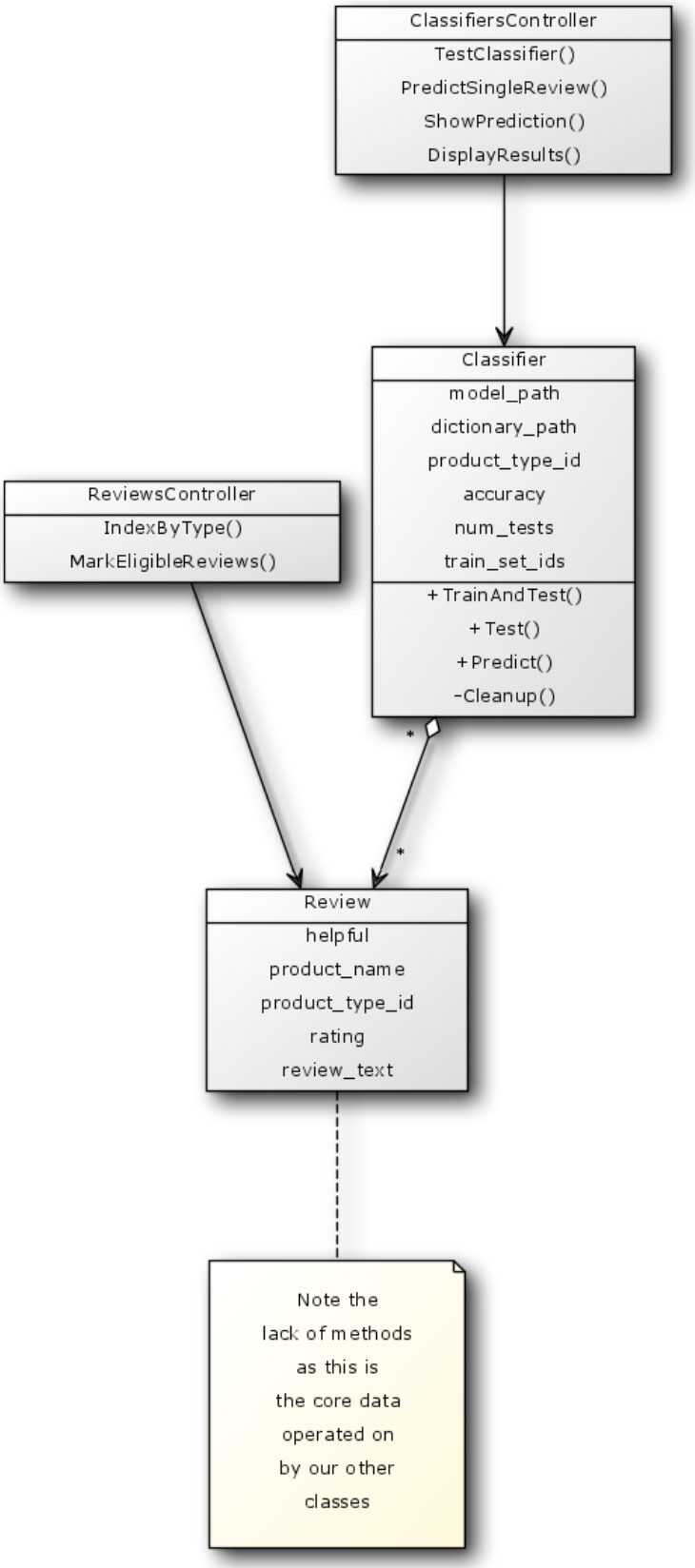
As the UML diagram shows, both reviews and classifiers belong to only one product type, but reviews also belong to a plurality of classifiers. Of course, each classifier uses only reviews that are from its same product type, but we can train multiple classifiers for a single product type and each classifier might contain overlapping sets of reviews used in other classifiers with the same product type for training or testing.

While not used throughout the entire application, the file upload model is an important part when compiling a product type's review set. We can append to this review set by importing many files, hence the product type's class aggregation of file

uploads. When uploading XML files containing reviews, however, all reviews are separated into distinct product type files, thus any upload belongs to only one product type, supporting reviews from multiple product types is not explicitly supported. The upload controller allows us to either process reviews from many XML files in a specified directory, or process each XML file individually. We further abstract the Nokogiri parser mentioned above into a function to retrieve the contents of a given XML node. As we process a file, a new review instance is created whenever we encounter encompassing *<review>...</review>* tags. Each attribute inside these tags is a node that must be processed and saved into a column of the record in the reviews database table.

The final class diagram, though simple, outlines the members doing the bulk of the work in the application. Although the file parsing and review organization into distinct product types and models is important, with nothing further, we would essentially have only built an Amazon.com review browser for 25 different product categories. Gorgeous though it is, there is no added utility over the Amazon.com website itself. Figure 8 displays the methods and relationships between the core functionalities of the web application.

Figure 8: Core Functionality Classes



It should be mentioned that both the classifier model and the review model contain additional attributes, but as they are either unused (in the case of the additional data contained in each review's XML nodes) or used only for housekeeping or helpers in the application (in the case of counts stored in the classifier model), they are omitted in this diagram. As can be seen from Figure 8, the review model is not modified after importing and organizing reviews into the database, but simply acted upon as core data. This diagram also shows a function, `MarkEligibleReviews`. The explanation of the ins and outs of this method are covered in a later section, but for now, it can be noted that after `MarkEligibleReviews` is executed on a set of reviews, those reviews are then ready for classifier training and testing.

Deployment and Schema

Thus far, we have identified the methods used to pull Amazon.com reviews from the Multi-Sentiment Dataset XML files into a MySQL database accessed by RHP, a Ruby on Rails application, defined the overarching design pattern comprising the web application, and outlined the relationships amongst application models. Lastly, we present RHP's deployment and database schema. The database contains three tables, classifiers, product_types, and reviews. Figure 9 illustrates the deployment of the application. Figure 10 presents the columns of each table.

Figure 9: Application Deployment

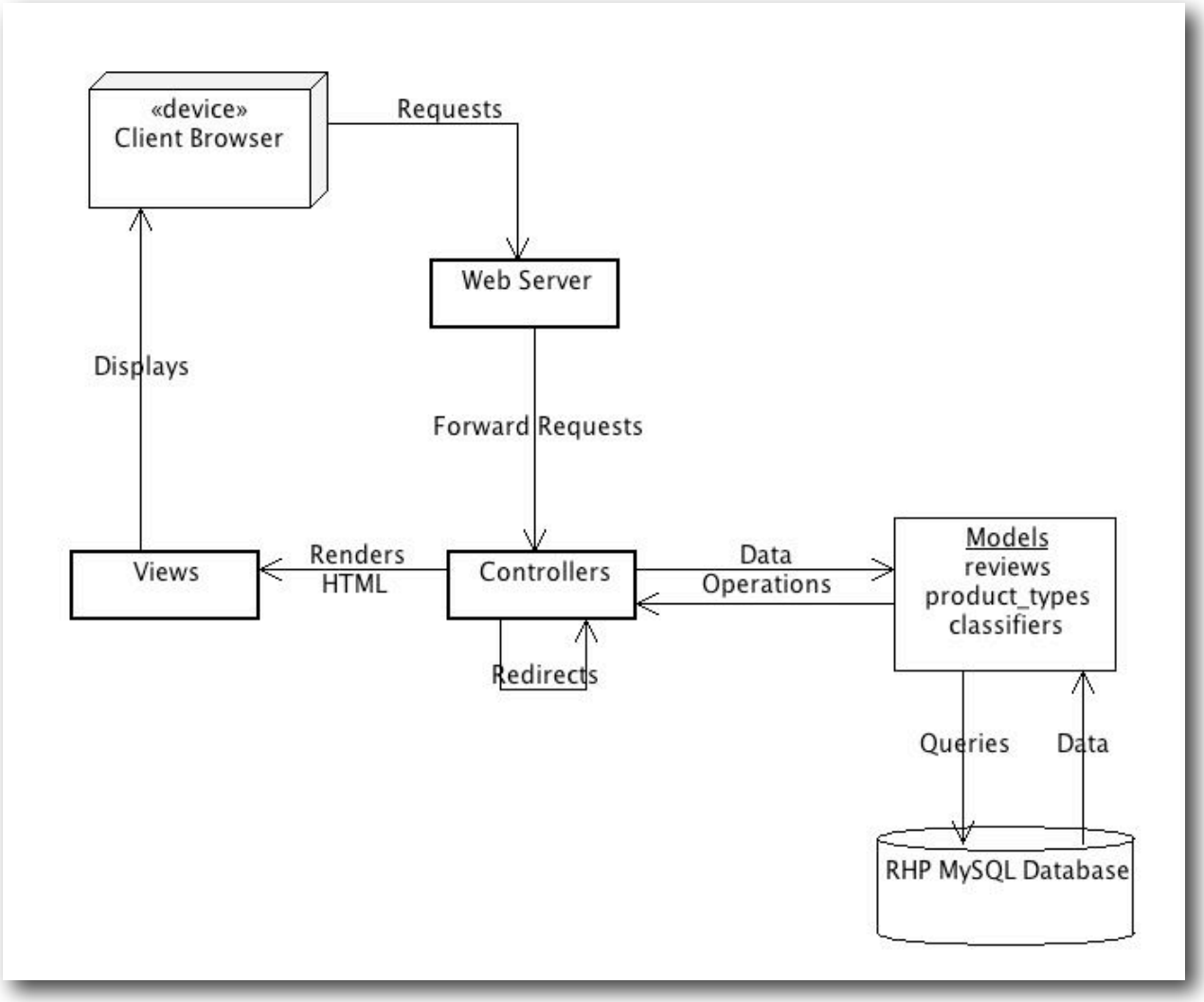


Figure 10: Database Schema

classifiers:

field	type	description
id	int	unique identifier
name	string	unique string based on the creation time, used for naming the persistent file names
model_path	string	full path to file storing the SVM classifier model data
accuracy	float	running average accuracy for all tests run
product_type_id	int	foreign key into the product_types table
created_at	date time	creation timestamp
updated_at	date time	last updated timestamp
dictionary_path	string	full path to file storing the global dictionary for this classifier
using_stopwords	boolean	boolean value stating whether or not stop words were removed from the global dictionary and training set
helpful_count	int	how may reviews used in the training set are helpful
unhelpful_count	int	how may reviews used in the training set are unhelpful
train_size	int	size of the training set
test_size	int	size of the test set
train_set_ids	text	list of all review IDs used to train this classifier
num_tests	int	running count of how many tests have been executed

product__types:

field	type	description
id	int	unique identifier
name	string	name of the product type
created_at	date time	creation timestamp
updated_at	date time	last updated timestamp
total	int	total number of reviews in this product type
helpful_total	int	total number of helpful reviews in this product type
unhelpful_total	int	total number of unhelpful reviews in this product type
eligible_count	int	total number of reviews eligible for training and testing use (flagged by `MarkEligibleReviews` function)

reviews:

field	type	description
id	int	unique identifier
product_name	string	name of the product being reviews
product_type_id	int	foreign key into product_types table
helpful	string	helpfulness voting string (ie: “4 of 5”)
rating	decimal	this review’s star rating for the product (1-5)
title	string	title of the review
date	date time	time stamp of when the review was created
reviewer	string	username of the Amazon user authoring the review
review_text	text	the complete review authored by the user
created_at	date time	time stamp of when the review was imported into database
updated_at	date time	time stamp of when the review was updated in database
is_helpful	boolean	flag to mark if more people voting on this review found it to be helpful than unhelpful
is_unhelpful	boolean	flag to mark if more people voting on this review found it to be unhelpful than helpful
degree	double	degree of agreement on helpfulness
use_for_train	boolean	flag to mark if the review is eligible for training and testing

As Figures 9 and 10 show, most of the work is done by the controllers and the models. The lightweight Rails server handles request forwarding, RESTful calls, and routing to the to the controllers, which interact with the models directly, and then render the views back to the browser in HTML.

The majority of the fields in the reviews table come from the XML nodes of the original data (outlined in Table 2) converted directly into MySQL data by our parser after files have been uploaded for processing. However, to aid in the simplicity of building the classifiers as well as the web page views, several columns were added to the database. Namely, the three boolean fields, `use_for_train`, `is_helpful`, and `is_unhelpful` were created for these reasons. As the names of the latter two suggest, these booleans are simply to very quickly determine if a review has more helpful than unhelpful votes or vice versa. Reviews that were not voted on will have both of these set to false. The third boolean column, `use_for_train` is explained in the next section.

Algorithms

This section outlines the algorithms used to take the review data we already have in the database, organize and partition it to choose training and testing sets, train SVM classifiers, and test and gather results. While the specific product type is not defined here, the steps are applicable to all product types in the dataset, with a few caveats that are explained in the results section.

Review Eligibility

As mentioned above, we have added some utility columns to the core data in the reviews table. In the research done by both Kim et al [8] and Hong et al [6] the authors set a “voting rate” of 0.6 to draw a boundary between helpful and unhelpful reviews, labeling the reviews that had been voted on beforehand, and then testing their classifiers on the entire dataset. We take a somewhat different approach and introduce the idea of review eligibility when compiling our training and testing sets. Eligibility is determined by the `use_for_train` column which is flagged as true or false using Algorithm 1 below.

Algorithm 1: Marking Review Eligibility

```
1  def mark_eligible_reviews( id )
2   for each r in all reviews with product_type_id = id
3    if r.helpful_votes < ( r.total_votes / 2 ) //unhelpful
4     r.degree = ( r.total_votes - r.helpful_votes) / r.total_votes
5    else //helpful
6     r.degree = r.helpful_votes / r.total_votes
7    end if
8    if r.degree > 0.7 && r.total_votes > 3
9     r.use_for_train = true
10   else
11    r.use_for_train = false
12   end if
13   r.save //persist to database
14  end for
15 end
```

This algorithm is run as a preprocessor step prior to training the classifiers. It is initiated with the click of a button located on the page where reviews are indexed according to product type. Once this button is clicked and this algorithm completes, we are left with a new partitioning of the product type's reviews. Where previously there were helpful, unhelpful, and reviews that had no votes, the review set is now organized as reviews with no votes, reviews with votes but without the required degree of agreement on their utility (helpful or unhelpful votes are too low or without a clear winner), and lastly, with a subset of reviews that are candidates for training and testing the classifiers. Note that in line 8, we stipulate that the degree of agreement on a review's utility (as voted by actual Amazon.com users) must be above 70 percent. This means that a review where 3 of 5 people found it to be helpful would be discarded as ineligible for classifier training because 40 percent of the users that took the time to vote found the review to be unhelpful. We also impose a minimum on the number of votes the review received. This criteria differs from past research as the above example review would simply be counted as helpful and used in the model. Our thinking is that reviews with a stronger degree of agreement, be it on the review's helpfulness or uselessness, will better inform our model during training and increase classification accuracy.

Selecting Training and Testing Sets

When using supervised learning to train models, it is important to select an appropriate training set. Ideally, this subset of data should be representative of the whole dataset to maximize the accuracy of the model. Initially, our algorithm was very

naive, selecting the training data from the database in the order that it was stored. For example, a request for 100 reviews in the training set and 50 reviews in the testing set would return the first 100 reviews that were eligible (as ordered by the id field of the reviews table, the unique, primary key) as the training set and then the next 50 eligible reviews as the testing set. This imposed a dependency on how the reviews were loaded into the database, as the reviews with lower primary key identifiers were favored in the training and testing of the model, which introduced problems. For example, if the first 99 eligible reviews happened to be helpful and the next 51 were unhelpful, our model would be plagued with overfitting and results would be unreliable. To mitigate this, next we tried to shape the query based on the product type's overall ratio of helpful to unhelpful reviews. For example, if there were 4,000 unhelpful reviews and 6,000 helpful reviews of grocery items, a request for the training set of size 100 reviews would return the first 40 unhelpful reviews and the first 60 helpful reviews. This approach is still problematic as it continues to favor one sector of the dataset.

The final implementation employs randomization to select a training and test set. This approach is outlined in Algorithm 2 below.

Algorithm 2: Selecting Training and Testing Sets

```
1  def generate_sets( train_size, test_size, id )
2   all_eligible_reviews <= get all reviews from database with
                            product_type_id = id &&
                            use_for_train = true
3   i = 0, train_reviews = {}
4   while i < train_size
5    random_review = all_eligible_reviews.sample
```

```
6    train_reviews.add( random_review )
7    i = i + 1
8    eligible_reviews.delete( random_review )
9   end while
10  j = 0, test_reviews = {}
11  while j < test_size
12   random_review = all_eligible_reviews.sample
13   test_reviews.add( random_review )
14   j = j + 1
15   eligible_reviews.delete( random_review )
16  end while
17 end
```

Here we employ the boolean flag `use_for_train` that was set in Algorithm 1 to obtain an entire set of all reviews of this product type that can be used for training and testing. Next, we make use of Ruby's `sample` method (lines 5 and 12), applicable to any collection of objects. This affords us the ability to pull an object, a review in our case, from the array using randomly generated indices. By deleting this review from the set of eligible reviews, we shrink this array each time we build up our training and testing sets, ensuring that we avoid duplicate entries in these collections. Note that this randomization yields the added benefit of a built-in adherence to the ratio of helpful to unhelpful reviews in the original dataset. Since we are randomly selecting each review for the training and testing sets, the probability that this selection is either helpful or unhelpful mirrors the entire dataset's overall statistics of helpful and unhelpful reviews.

Building a Classifier

The training of a model in machine learning can be simplified into three entities: input (training data), learning process, and output (a model ready to accept data for classification). The previous section explains how we derive the first entity. Below, Algorithm 3 outlines how we use the second entity, the learning process, to output the model.

Algorithm 3: Training SVM Classifier

```
1  def train_classifier( training_set, use_stopwords )
2   documents = {}
3   for each r in training_set
4    if r.is_helpful
5     documents.add( [ 1, r.review_text ] )
6    else
7     documents.add( [ 0, r.review_text ] )
8    end if
9   end for
10  global_dictionary  = all unique words in all of documents
11  global_dictionary = global_dictionary stripped of punctuation
12  if use_stopwords = true
13   read each stop word from stopwords file and delete from global
     dictionary
14  end if
15  stem all words in global dictionary
16  training_vectors = {}
17  for each d in documents
```

```
18    strip out punctuation of d
19    training_vectors.add( d.label, features vector )
20  end for
21  problem = new LIBSVM problem
22  parameter = new LIBSVM parameter //using proven defaults
23  parameter.cache_size = 1
24  parameter.eps = 0.001
25  parameter.c = 10
26  problem.set_examples( training_vectors )
27  model = LIBVM::Model.train( problem, parameter )
28 end
```

At the end of Algorithm 3, we have a model that is ready to accept and classify reviews from the test set. This algorithm relies heavily on an implementation of LIBSVM installed as a Ruby plugin. LIBSVM is an effective and efficient implementation of SVM classification. It was developed by Chih-Chung Chang and Chih-Jen Lin [7]. The web page, http://www.csie.ntu.edu.tw/~cjlin/libsvm/, offers extensive information about this software and links to other language plugins. The Ruby plugin used, rb-lisvm, available at https://github.com/febeling/rb-libsvm, is a Ruby binding to this LIBSVM software. In addition to LIBSVM, we also employ a word stemming plugin from Roman Shterenzon, fast-stemmer. It is available at https://github.com/romanbsd/fast-stemmer. This allows us to efficiently stem words in our global dictionary and each training document.

After organizing our documents into a set of training vectors mapped against each feature (word) in our global dictionary, we use the LIBSVM class to train the

model. The training function accepts the vectors and a parameter object as input. The parameter object stipulates important details about the SVM classifier. These details are explained in depth by Chang et al [7] and this algorithm adopts the parameters that have been proven to work well in similar classification problems.

While not shown in the algorithm, it is worth mentioning that the LIBSVM model contains a method to save its data to a file. This, coupled with our implementation to save the global dictionary used to train the model, allows the application to easily reload any trained classifier for future testing.

Testing a Classifier

With the models trained and saved as outlined in Algorithm 3, we can load and feed in a test set to observe the classifier's accuracy. In machine learning, to predict accuracy of a model, the test set must be comprised of data with known classes. That is, our reviews in the test set are reviews that have already been voted to be either helpful or unhelpful. Recall from Algorithm 2 that we employ the same flag used for marking training reviews as eligible to create a pool of eligible test reviews. Algorithm 4 describes how classifiers are tested.

Algorithm 4: Testing SVM Classifier

```
1  def test_classifier( classifier_id, test_set )
2   helpful_total = unhelpful_total =
    helpful_correct = unhelpful_correct =
    correct = incorrect = total = 0
3   c = Classifier.find( classifier_id )
4   dictionary = File.read( c.dictionary_path )
```

```
5  model = LIBSVM::Model.load( c.model_path )
6  for each r in test_set
7   actual = -1
8   if r.is_helpful
9    actual = 1
10   helpful_total += 1
11  else
12   actual = 0
13   unhelpful_total += 1
14  end if
15  if actual != -1
16   strip out punctuation of r
17   r_features = vector created by checking stemmed words of
                  r.review_text against model dictionary
18   prediction = model.predict(LIBSVM::Node.features(r_features))
19   if prediction == actual
20    correct += 1
21    if actual == 1
22     helpful_correct += 1
23    else
24     unhelpful_correct += 1
25    end if
26   else
27    incorrect += 1
28   end if
29   total += 1
30  end if
```

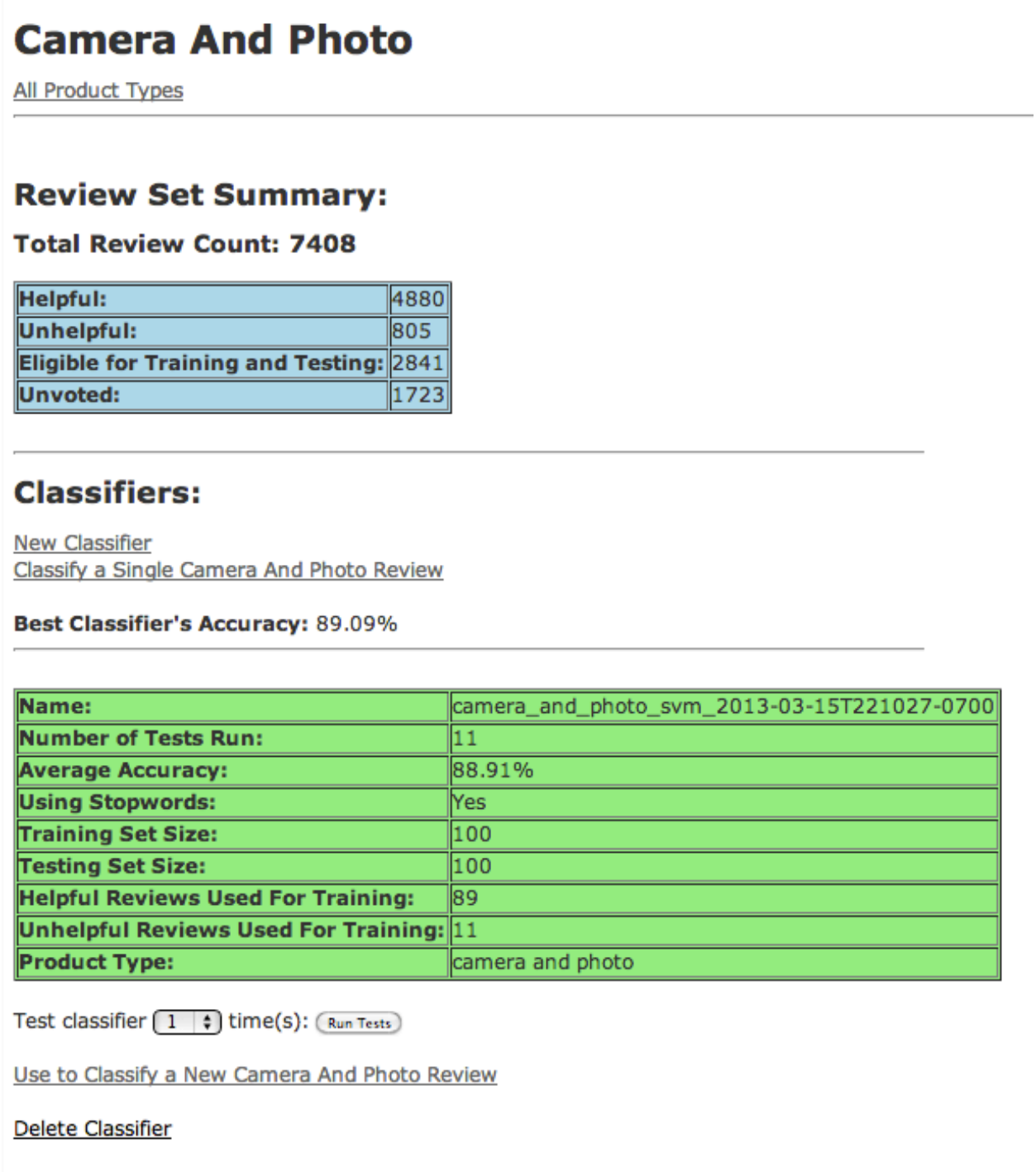
```
31   end for
32   computer new average accuracy for c
33   c.save
34   return results of testing
35 end
```

This algorithm is fairly straightforward, with the main point being the comparison of the model's prediction of the review against the known class (helpful or unhelpful) of the review. Also of note is the responsibility of a classifier to keep track of its accuracy. This allows us to create competing classifiers for the same product category to track which is the best performing. Tracking the classifier with the highest accuracy affords the application the ability to chooses which classifier is to be used to predict the utility of a review that is entered on-the-fly by a user, explained in the next section. Lastly we are returning the results of testing to the application's classifier controller, to render a view with the results presented on a web page.

Predicting a Single Review

While the testing algorithm (Algorithm 4) above encompasses predicting a single review as well (executed with a user-entered review as the sole member of the test_set), we explain this framework explicitly as, to our knowledge, it has yet to be suggested in previous work. The screen shot in Figure 10 shows what an indexing of classifiers page looks like for the "camera and photo" product type.

Figure 11: Classifier Index for Product Type



Camera And Photo

All Product Types

Review Set Summary:

Total Review Count: 7408

Helpful:	4880
Unhelpful:	805
Eligible for Training and Testing:	2841
Unvoted:	1723

Classifiers:

New Classifier
Classify a Single Camera And Photo Review

Best Classifier's Accuracy: 89.09%

Name:	camera_and_photo_svm_2013-03-15T221027-0700
Number of Tests Run:	11
Average Accuracy:	88.91%
Using Stopwords:	Yes
Training Set Size:	100
Testing Set Size:	100
Helpful Reviews Used For Training:	89
Unhelpful Reviews Used For Training:	11
Product Type:	camera and photo

Test classifier 1 time(s): Run Tests

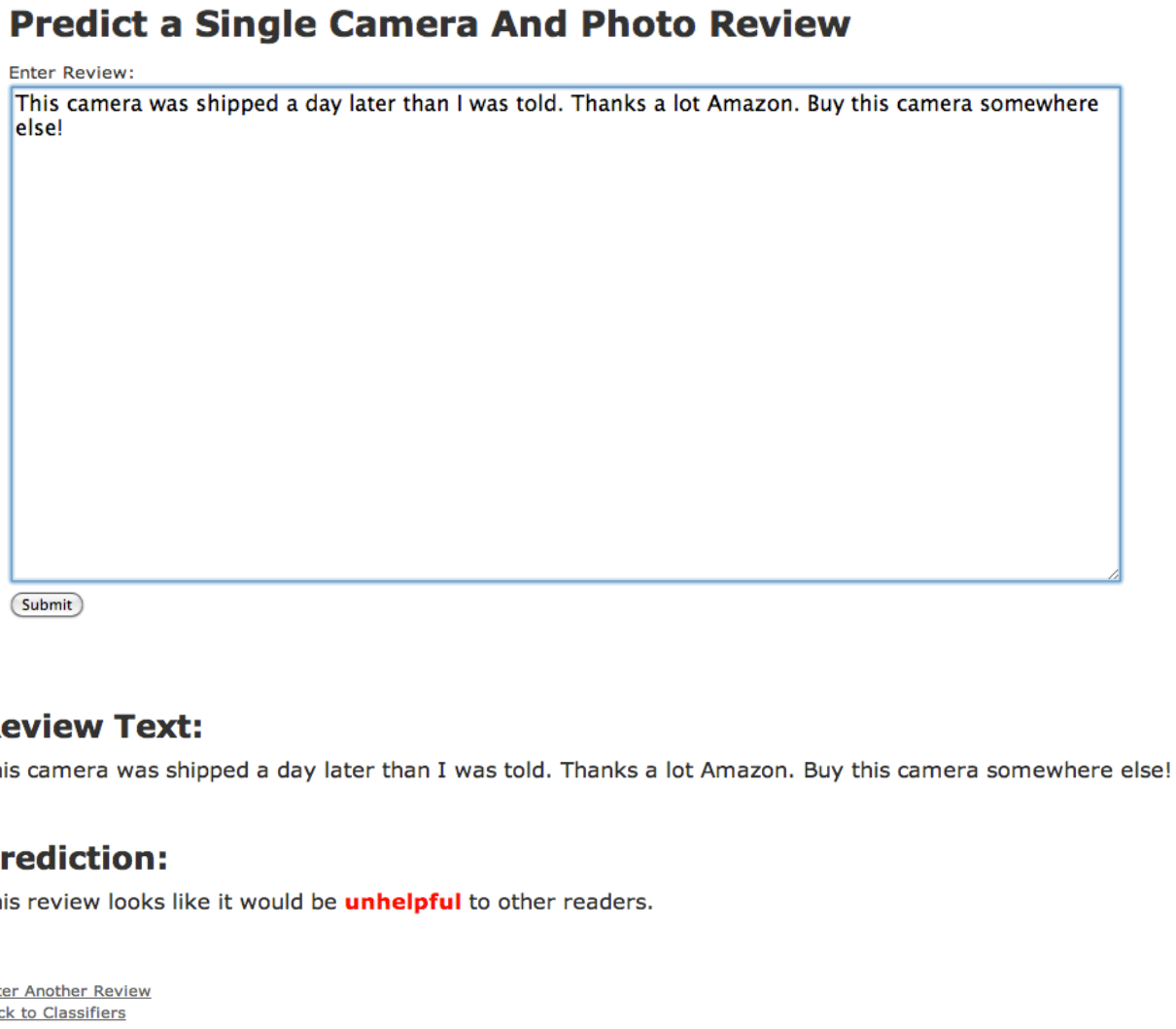
Use to Classify a New Camera And Photo Review

Delete Classifier

The green table holds information about only one of three classifiers that have been trained for this product category. Indeed, there are two others below that are not shown, one of which is the most accurate, with an accuracy of 89.09%. Although we

have built in links to use each of the classifiers to predict a single, user-entered review, the link at the top automatically uses the classifier with the highest accuracy. The process is shown in Figure 11 below.

Figure 12: Predicting a Single New Review



As this review is concerned only with the vendor and the shipping satisfaction of the author, we would suspect it would have little utility to other readers looking for information about the product itself. Our classifier's prediction agrees, marking this

review as unhelpful. While simple in nature, we believe that this framework could be implemented in online shopping websites and would help users to author higher quality reviews.

Results

Both Kim et al [8] and Hong et al [6] also used the Multi-Domain Sentiment Dataset from Blitzer et al [3] for results testing, but as the latter research outperformed the former, we use the results of Hong et al as a baseline for our comparison. Table 3 presents the results of past research.

Table 3 - Past Research Classifier Accuracy

Research	Accuracy (%)
Kim	61.29
Liu	62.85
Hong	69.62

While each evolution of this research has improved classification accuracy, all research thus far has, to our knowledge, focused on lumped-together product type reviews. As shown in Table 4 by the accuracies of our best classifiers for each product type, there are gains to be had by specializing the models for specific partitions of the review dataset.

Table 4 - Specialized Review Helpfulness Predictor Results

Product Category	Best Accuracy (%)
apparel	90.82
automotive	81.82
baby	92.0
beauty	93.55
books	81.91
camera & photo	89.09
cell phones & service	85.0
computer & video games	90.55
dvd	79.18
electronics	86.73
gourmet food	90.73
grocery	88.36
health & personal care	90.55
jewelry & watches	89.0
kitchen & housewares	93.45
magazines	85.82
music	77.0
musical instruments	93.82
office products	95.0
outdoor living	90.82
software	85.91
sports & outdoors	89.36
tools & hardware	100.0
toys & games	90.18
video	80.64

For each product type, we have trained three classifiers with training set sizes of 100, 300, and 600 product reviews wherever possible. As the table shows, our

accuracies for the Amazon.com product reviews are considerably higher than previous work, with some caveats. In the “tools & hardware” category, for example, only 40 reviews met our criteria for training and testing eligibility. Among these, only a small number were voted unhelpful. Whereas most product types have classifiers with a minimum training size of 100 reviews, here we are only able to train the model with 20 reviews, leaving 20 for testing. This results in overfitting our model and without a wide array of test reviews available, the accuracy is artificially high. While this anomaly is present in a few other product types that are less popular areas for consumer interest on Amazon.com, this artificially high accuracy seems to be the exception. Even the most popular categories such as books, DVDs, and music, all have classifiers that perform quite well in comparison to past research. It is also clear that the most specific product categories tend to have the higher accuracies. For example, while the camera and photo product type could actually be considered “electronics” and lumped into this product type, the fact that it is separate and houses distinct, more specific reviews, lends itself to RHP’s specialized models. This is what one would expect as the dictionary for electronics is far broader than any sub-category of electronic products might be, thus increasing the challenge of accurate classification.

Conclusions

In this research we incorporated a dataset from XML into a more manageable and efficient format in a MySQL database. Using queues from past research, we pursued Support Vector Machines as an avenue for machine learning to automatically predict review helpfulness. Using Ruby on Rails, employing a Model-View-Controller

architecture, and an implementation of LIBSVM, we built the Review Helpfulness Predictor (RHP) web application with functionality to train and test models, and predict single product reviews as either helpful or unhelpful. As proven by our results, our hypothesis of creating specialized machine learners to increase performance on a partitioned dataset is an effective way to classify product reviews. Aside from these improved results, we also offer a framework for selecting and employing a classifier to perform on-the-fly classification of product reviews entered by customers, with the end goal of improving review quality to the benefit of customers, online vendors, and manufacturers.

Future Work and Direction

As our hypothesis appears valid, we see no reason why machine learners should not be even further specialized by partitioning the product types further. Given adequate numbers of reviews for single products, it is not inconceivable to have a classifier trained specifically for one popular product sold. We also suspect that incorporating more words into a single feature (using bigrams, trigrams, quadgrams, etc.) during training, rather than treating each word as a distinct feature might have a positive effect on performance. For example, we treating the presence of “focal length” as one feature in a ‘camera and photo’ review rather than “focal” and “length” as distinct features.

MAKifiers

As part of a further analysis of the relationship between kinds of product reviews used for training versus the kind of product review we are predicting as helpful or unhelpful, we have created an additional model type, the MAKifier (Many Applicable

Kinds) classifier. As opposed to the classifier model, a MAKifier can have many kinds of product reviews used for the training set, and is used to predict the helpfulness of reviews from one kind of product. This affords us a sliding scale from a highly specialized machine learner to the type of classifier used by previous research (one classifier for all kinds of product reviews). Our preliminary testing included using all 25 product types to train a model and then test it against only ‘camera and photo’ reviews. Not surprisingly, this yielded an accuracy close to, but slightly less, than previous research, 60.7%. We also tested ‘camera and photo’ reviews against classifiers that were trained with fairly unrelated kinds of products, ‘baby’, ‘dvd’, ‘office products’, ‘beauty’, ‘magazines’, and ‘health and personal care’. These tests yielded accuracies of 52% and 55%, only slightly higher than simple chance. One interesting observation from early MAKifier testing was the improved accuracy when training reviews are used from a kind of product that is similar to the one used for testing. For example, when a learner was trained with ‘dvd’ and ‘video’ reviews and was tested against ‘book’ reviews, the accuracy was significantly higher at 68% as opposed to testing ‘book’ reviews against ‘apparel’, ‘automotive’, and ‘kitchen and housewares’ reviews, where accuracy was only 51%. We suspect this is due to the common elements of reviews from ‘dvd’, ‘video’, and ‘books’ such as story, emotion evocation, and prose commentary. This warrants further research to extrapolate indicators of similarities between different kinds of products, as well as the relationship between the number of product types used for training a model versus the number of product types used for testing.

Lastly, we believe that the interface for predicting a single review and reporting the predicted helpfulness to the author could benefit from added details. For example,

rather than simply stating whether their review looks to be either helpful or unhelpful to other readers, enhancing the feedback to include specific content that could be added or improved upon would ultimately guide the author toward creating a review of higher quality and helpfulness.

References

[1] Mumtaz M. Al-Mukhtar and Yasmine M. Tabra. 2012. An effective spam filter based on a combined support vector machine approach. Int. J. Internet Technol. Secur. Syst. 4, 1 (January 2012), 42-54. DOI=10.1504/IJITST.2012.045149 http://dx.doi.org/10.1504/IJITST.2012.045149

[2] Izzat Alsmadi, Mohammed Al-Kabi, Abdullah Wahbeh, Qasem Al-Radaideh, and Emad Al-Shawakfa. 2011. The Effect of Stemming on Arabic Text Classification: An Empirical Study. Int. J. Inf. Retr. Res. 1, 3 (July 2011), 54-70. DOI=10.4018/IJIRR.2011070104 http://dx.doi.org/10.4018/IJIRR.2011070104

[3] John Blitzer, Mark Dredze, Fernando Pereira. Biographies, Bollywood, Boom-boxes and Blenders: Domain Adaptation for Sentiment Classification. Association of Computational Linguistics (ACL), 2007

[4] Rong-En Fan, Pai-Hsuen Chen, and Chih-Jen Lin. 2005. Working Set Selection Using Second Order Information for Training Support Vector Machines. J. Mach. Learn. Res. 6 (December 2005), 1889-1918.

[5] Anindya Ghose and Panagiotis G. Ipeirotis. 2007. Designing novel review ranking systems: predicting the usefulness and impact of reviews. In Proceedings of the ninth international conference on Electronic commerce (ICEC '07). ACM, New York, NY, USA, 303-310. DOI=10.1145/1282100.1282158 http://doi.acm.org/10.1145/1282100.1282158

[6] Yu Hong, Jun Lu, Jianmin Yao, Qiaoming Zhu, and Guodong Zhou. 2012. What reviews are satisfactory: novel features for automatic helpfulness voting. In Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval (SIGIR'12). ACM, New York, NY, USA, 495-504. DOI=10.1145/2348283.2348351 http://doi.acm.org/10.1145/2348283.2348351

[7] Chih-Chung Chang and Chih-Jen Lin, LIBSVM : a library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2:27:1--27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm

[8] Soo-Min Kim, Patrick Pantel, Tim Chklovski, and Marco Pennacchiotti. 2006. Automatically assessing review helpfulness. In Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing (EMNLP '06). Association for Computational Linguistics, Stroudsburg, PA, USA, 423-430.

[9] Raymond Y. K. Lau, S. Y. Liao, Ron Chi-Wai Kwok, Kaiquan Xu, Yunqing Xia, and Yuefeng Li. 2012. Text mining and probabilistic language modeling for online review spam detection. ACM Trans. Manage. Inf. Syst. 2, 4, Article 25 (January 2012), 30 pages. DOI=10.1145/2070710.2070716 http://doi.acm.org/10.1145/2070710.2070716

[10] Ying Liu, Jian Jin, Ping Ji, Jenny A. Harding,and Richard Y. K. Fung. 2013. Identifying helpful online reviews: A product designer's perspective. Comput. Aided Des. 45, 2 (February 2013), 180-194. DOI=10.1016/j.cad.2012.07.008 http://dx.doi.org/10.1016 j.cad.2012.07.008

[11] Thomas L. Ngo-Ye and Atish P. Sinha. 2012. Analyzing Online Review Helpfulness Using a Regressional ReliefF-Enhanced Text Mining Method. ACM Trans. Manage. Inf. Syst. 3, 2, Article 10 (July 2012), 20 pages. DOI=10.1145/2229156.2229158 http://doi.acm.org/10.1145/2229156.2229158

[12] Michael P. O'Mahony and Barry Smyth. 2010. Using readability tests to predict helpful product reviews. In Adaptivity, Personalization and Fusion of Heterogeneous Information (RIAO '10). LE CENTRE DE HAUTES ETUDES INTERNATIONALES D'INFORMATIQUE DOCUMENTAIRE, Paris, France, France, 164-167

[13] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. 2005. Introduction to Data Mining, (First Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.