CLASSIFICATION OF TEXT DOCUMENTS USING DOCUMENT CONTENTS

By

Daniel Wojcik

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Approved:

_____

Mukkai Krishnamoorthy, Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

July 2009
(For Graduation August 2009)

CONTENTS

LIST OF TABLES

ACKNOWLEDGMENT

I would like to acknowledge the support of my advisor, Mukkai Krishnamoorthy. His advice and encouragement has always been available when needed, both on this thesis and my academic career in general. His insight has been invaluable and is greatly appreciated.

I would also like to thank the team at Google Books for providing the data we used to test our program. Having real world data to use, with all the idiosyncrasies that brings with it, was a great help to gauging the practicality of our results.

I would further like to thank Sean O'Sullivan and the Rensselaer Center for Open Source Software for sponsoring my final months here at RPI. The funding provided enabled me to rest easy and focus on my work.

Finally, special mention is needed for my family. Their boundless support for everything I've tried to accomplish throughout my time here has been extraordinary.

ABSTRACT

Proper organization of documents is an important operation in many fields. Looking specifically at text documents, many of which would be produced by OCR data of questionable accuracy, we test the viability of several different semi-supervised statistical classification methods. To achieve this, we implemented an extendable program with many adjustable parameters to allow significant testing over wide ranges of values and algorithms. Looking at four specific schemes and their associated parameters, we examine the successes and failure of their usefulness on a database of one thousand books from the late seventeenth to nineteenth centuries. While the results found were not as accurate as we had hoped, we discuss the feasibility and related issues of this approach along with possible extensions and further applications of our implementation which may improve performance.

1. INTRODUCTION

1.1 Statement of Problem

In any field, proper organization of documents is important in order to ensure the information contained within them is easily accessible. The systems used to categorize documents vary by situation — from a simple alphabetical list of patients in a doctor's office to the Dewy Decimal System employed by libraries to the meta tagging used in web content. All of these systems are generally straightforward for a person applying them to a single document as an ongoing iterative addition to an existing database.

However, when starting from scratch on a completely un-categorized database, the man hours required to determine the necessary information for each document can be prohibitively high. Thus, an automated solution to categorizing documents based on their actual content instead of secondary tags would be extremely helpful. Such a system could reduce the human role to that of overseer, as well as provide support for people with visual impairments that would make manual classification difficult.

1.2 Previous Work and Approach to Solution

Statistical classification is a very prevalent operation, appearing in everything from spam filters to text searches to image recognition. To that end, this thesis is not intended to revolutionize the field with an entirely novel approach to classification. Instead, it focuses on a specific problem where these methods could be useful, and attempts to determine the feasibility and issues of the approach. By implementing versions of existing methods in an easily extendable manner, we hope to be able to detail the successes and failures of these algorithms with regards to this particular area, while leaving something useful behind that further researchers could build upon to refine the results.

As noted by Halevy et al., simple statistical measures can be extremely effective in comparison to more complex methods, given enough data[1]. Perhaps even

more importantly, they suggest that such measures are less likely to be thrown off by errors in the data — misspellings, incomplete sentences, etc.. This was an important property as our test database, and intended field of use should it work, consists primarily of scanned document pages. Optical character recognition, particularly on the older documents in our set, is notoriously inexact, leading to vast numbers of misspelled or otherwise broken words and phrases[2]. Although our own data set is very small in relation to the corpus they were referring to (i.e., the internet), we wanted it to be scalable to these levels. Specifically, if this were to be run on an several library's worth of documents, it should follow this same principle. To that end, we sought purely statistical classification schemes of varying, but moderate, complexity. Our choices will be discussed in more detail later, but we picked four methods to implement. The first is a very basic weighted means approach. The second extends that to include one of the simplest clustering schemes available, k-nearest neighbors[3]. The third is the more modern and very popular support vector machines[3]. The fourth is a somewhat different style of method, though still a statistical comparison, using Jaccard Coefficients[4]. We also chose to implement each of these as semi-supervised learning algorithms, as these can provide significant improvement over purely unsupervised methods, but manual classification of large numbers of documents would be prohibitive in a practical setting[5]. Specifically, our implementations will integrate the unlabeled data that they classify back into their own training sets for future use.

To achieve this end, we have developed a learning algorithm to perform classification using these methods. At the most basic level, our system takes two data sets: the first is a manually classified sample, and the second is the database for automatic categorization. The result is a classified database in accords with the scheme used in defining the manual set.

The reasons for the training set are twofold. First, this provides a more intuitive method of specifying the desired categories and scheme. Different situations could have wildly different kinds of classes — date, subject, even language — and allowing the users to specify those without having to deal with arcane notation in configuration files or direct source code will aid accessibility. Second, we decided

that semantic and syntactic analysis is beyond the scope of this thesis, and thus the algorithm needs some baseline for comparison of pure token analysis. We accept the influence the specific selection of this set can have on the end result, and some of the more complex methods detailed later can mitigate the impact of a poorly chosen training set.

For the operation of the system we had two main objectives, speed and versatility. The algorithm needs to be implemented efficiently enough to achieve reasonable running times, or the automated system will not be useful in a practical setting. Versatility, on the other hand, is important because of the aforementioned variance in category types. We suspected that certain kinds of algorithms may be better suited for specific class types — for example, one may work better at determining a content subject, but another could be more accurate at finding language cues that suggest certain time periods. To this end, we designed the main system to allow different methods for classification and clustering to be implemented and plugged in with potentially little adjustment to the rest of the code depending on their needs and operations. Many tweaks are also handled as separate parameters that can be chosen by users to fine tune the results. Both of these operations also aid in testing — we tried multiple different methods ourselves, of varying complexity and power, the specifications and results of which are shown later in this thesis.

2. METHODS AND IMPLEMENTATIONS

2.1 Underlying Program Method

As mentioned in the previous section, the program we developed takes two input sets — one with manually defined classes and one without — and will classify the second set. Thus, it has two basic modes, one where it reads pre-classified documents and one where it reads un-classified documents and assigns classes to them. These two steps must be run separately, though a simple script can be used to execute them in sequence.

The core operation is identical in both cases, however, since that is just the act of reading and analyzing the contents of each document. As noted above, no semantic analysis is used, and even syntactic analysis is skipped — all of the information it gains is based solely on token counts. This provides more versatility in the core system, as semantic and syntactic rules would need to be determined for different languages and contexts. It is worth exploring ways such information could be calculated automatically in the learning stage, but such an extension is beyond the scope of the work in this thesis, and touched on in the future work section.

Starting off, the program header file defines some of the operational parameters used to tweak the behavior of the system. Specifically, the bounds on what is considered to be a relevant term (as discussed later in this section) are defined here, as well as size limits of the characteristic terms and assigned classifications, and some algorithm selection constants. Further parameters are provided as command line arguments to the program. Specifically, these provide a flag indicating whether it is in learning or classification mode, and the input file. They also allow a flag to specify not using a saved database and the rate of characterization (discussed later). The input files are actually lists of filenames for the program to load, as this makes calling the program on large data sets much simpler.

Several classes and structures are also defined to store information at the various stages and levels of the program, and are provided in the DocItem file and its header. Three structures are used for storing term information — one for use at

the document level, one for use at the class level, and one for the global level of the program. The document related version stores an identifier and a occurrence count. The global level keeps track of the total occurrences in all documents, the number of documents it appears in, and the *inverse document frequency* (idf) of the term. The idf is a measure of the relative number of documents the term appears in (see B.1)[3]. This structure also holds a class count map, keyed to the class id strings, containing the number of occurrences of the term in each specific class. The class level, on the other hand, keeps track of the occurrences in all documents of that class, the number of documents (again, of that class) the term appears in, and the idf. These allow for the structures and containers tracking higher level information (e.g., the document or a class of documents) to hold the most relevant term information for their own operations.

In that vein, document information is stored in a class, which contains a map — keyed to the parsed string of each term — to the document term item. It also holds a count of all terms it has seen and two arrays (of a number of elements specified in the header defines). These represent the classes it has been sorted into as well as the classes it should be in. In addition, it has methods to set, increment, and get the occurrence count of a specific term in the document. Class data, on the other hand, is a structure with counts of the number of documents and terms in the class. It also contains an id of any cluster it is assigned to, and a boolean to keep track of proper setup. Additionally, it uses a map, keyed to term strings, to house the aforementioned *characteristic terms* of the document (as explained later in this section), and a count of the number of terms contained within. Finally, it has a few algorithm related pieces of information that will be detailed in their appropriate sections.

Global containers and values are housed in a separate globals class, in order to allow access to this information in different program files (explained more below). Specifically, it uses three hash maps to house the global term, class, and clusters (clusters discussed in section 2.2.2). These are keyed to parsed strings for terms, id strings for classes, and id numbers for clusters. It also includes global counts of the total number of documents and terms seen. Term information makes use of

the standard library maps here, as it does in the other areas it is used, in order to facilitate random access of the information. Since the access pattern depends on the order of terms in the document being read, or the chosen subset labeled the characteristic terms, being able to grab a term anywhere in the container to access or update its information is important. It is more common for the class and cluster maps to be iterated over, generally to compute related information or compare against each one. However, they also involve frequent random access when updating their own information, so the map was still used.

The actual program code is divided into two files, one for main and one for algorithms (with algorithms containing an additional header file). This is to separate the method-specific details from the core program operation. In other words, the main program can call a function, and it shouldn't matter exactly what that function does, as long as the relevant information is tracked and returned. This means that all methods are implemented within the functions called by main, and housed within the algorithms file. Naturally, however, there is often enough disconnect between what different approaches need to do that it is unwieldy to try and include them in the same function. For that reason, some of the functions called are mere dispatchers — essentially switch statements that use the globally defined metric parameter to call a function specific to the current algorithm. Details of some of these will be provided in discussion of the core operation below, and the rest will be given in the sections of their respective methods.

To start off, however, the algorithms file contains a very important set of functions that are used repeatedly throughout all of the methods. Specifically, this is the scoring function for a term, applied to the different versions of the term structures. As this is a highly prevalent operation and is used in all of the algorithms in some capacity, changing it can have significant impacts on the system results. For the purposes of this chapter, the scoring method is the product of a constant scalar (defined in the header parameters) multiplied by the normalized frequency-idf value of the term. This value is the term idf multiplied by the term count, and then divided by the number of documents it appears in. This formula is provided in B.2, but it is worth noting that other versions of it will be discussed in later chapters.

The main program begins, after processing command line arguments, by loading any stored knowledge it is provided. The specifics of the information stored are mentioned later in discussion of saving them, but after they are loaded, the algorithm dependent information must be computed by performing the clustering action (also discussed later). For now, it is enough to say that it reestablishes its global information and any decision criteria from saved files, and can then proceed with reading the input file. Each line should contain a filename. Once it has opened a document, in order to perform its pure token frequency based analysis, the system creates an object for the document, and reads through the file. It keeps track of all occurrences of each term, updating the count in the document, the global count for that term in all documents seen so far, and the term idf.

The system then determines the true class of this document based on meta data provided in the file. In a practical setting, this would not be available. However, we made use of this information so that the program could keep track of its own accuracy. This also eliminated the need for class labels to be provided in the input file of the training set. Regardless, when in classification mode, our program will then call a dispatcher function (classify) to set an algorithmically chosen class as the actual assigned class. In learning mode the assigned class will simply be the true class. It will then update the global term idf information for all seen terms, as well as increase the term class counts for the class this document is now part of. Additionally, it updates the document count of the class the new document has been sorted into. Continuing on, the program will then potentially compute some additional information based on the rate provided in the arguments — it will perform these steps after every X documents are read, in order to provide some control on the time associated with these functions — with the default being 16. More directly, the steps included here are characterizing the classes, clustering, and saving the current global information to files.

The first action, characterizing, is provided in algorithms but is not a dispatched function. Instead, it computes the characteristic files of each class, which are the terms it decides to be the most relevant as far as distinguishing power is concerned. This is a form of feature reduction, but they are not necessarily the same

for each class. Thus, they provide a heuristic for classification that does not involve comparisons over every single term seen. Like most of our methods, we strove for a simple online measure. Thus, we used a measure similar to the *probability ratio* discussed by Li and Sun as a component of their Scalable Term Selection method[6]. The max number of characteristic terms is a globally defined parameter to allow user control over the speed and accuracy tradeoff of the heuristic. This function starts off by weeding out terms deemed irrelevant — those which are too infrequent to be statistically significant (low global count) or too common in different documents to contain any distinguishing power (low idf)[3]. This is done using bounds defined in the header file, scaled according to the number of documents seen so far (see B.3). Then, it iterates over all classes, and for each class over all globally seen terms. It applies a score function to the term's data for that class (see above), and then finds the lowest score among all currently chosen characteristic terms. If the score is higher than this, that minimum term will be replaced by the new one. These characteristic terms are then stored in the structure for its class. This is usually one of the longest single computation steps, as it involves calculations over all classes and all terms. However, since it results in smaller working sets for future term computations, it can reduce the time spent in other areas.

Once the classes are updated, the system will attempt to cluster them in the second action. This is another dispatching function, and is perhaps unfortunately named as it does not necessarily involve clustering in every method. Instead, these functions involve recalculating the decision criteria for classifying new documents. This can be slow as well, depending on the specifics of the operations, and is thus separated out along with characterizing. The details of these functions are provided in the sections of their respective methods.

The remaining actions are simply to save the global information to files, so that future runs of the system can make use of the knowledge already accumulated in operation. This information is actually stored in two files — one for terms and one for classes — to keep the information highly targeted to the necessary data structures. Essentially, the files correspond to the global term and class structures respectively. Individual document information is irrelevant at this stage, and any

other required information is method dependent and thus would be computed after loading the files. This allows for training and classification to be separate executions, but also allows it to function similarly to how current schemes are used — once the initial database is categorized, incoming documents must be classified and added to the set. By saving and loading the necessary information, the system can be used iteratively as well as on complete sets, making it practical for continued use beyond the initial categorization. On a robustness note, it also means that if an execution should be aborted for whatever reason — power failure, system crash, etc. – the partial results are saved and execution can be started from the first document after the last save.

Once this is finished (or skipped), the program will move on to the next document until it reaches the end of the input file. Once that happens, it will again perform the characterizing and clustering steps in order to have the most up to date information to save to its knowledge files. Upon doing so, the program operations end.

2.2 Algorithm Implementations

2.2.1 Weighted Means

The first classification method implemented is a simple weighted means comparison. At the highest level, this compares the document properties with the stored class properties, applies a weighted scoring metric, and picks the one with the best result. As suggested by the nature of the method, the implementation is also very straight forward. It does not actually involve any operations in the clustering action, instead performing classification based solely on the general information stored for every class and document. Despite this simple approach, it provides a useful baseline and quick operational speed.

Given a document object, it will iterate over all classes in the map, and for each class iterate through its characteristic terms. For each term it will determine a difference score from the properties of that term in the class and the document, and sum these up to determine the weighted score for that class. It also keeps track of the highest score it has seen so far, and once it has gone through all classes, the

one with the highest score is chosen as the class of the document.

The weight function used is a function of the counts and idf of the term, similar to the term score used in determining the characteristic terms. The term count in the chosen document is first normalized by dividing it by the number of documents in that class, to determine a mean frequency of the term in this class. Then the absolute value of the difference between this mean count and the term's count in the provided document is multiplied by the term's global idf and a constant scalar. Summing this over all characteristic terms in the class will provide the score of the class. See B.4 for the formula.

2.2.2 Clustering

The second classification method is an extension of the weighted means to incorporate clustering into the decision process. It makes use of a nearest neighbor approach to group similar classes together. While k-nearest neighbors can be used as a classification scheme by choosing the class based on the classes of the neighbors[3], here it is used to provide basic clustering and secondary class information on documents.

The clustering action, therefore, first needs to determine the nearest K classes to each class. This number can be defined in the header parameters. This is done with the simplest implementation, a pair-wise comparison between all pairs of classes to calculate a distance metric between them. It keeps track of the smallest K values for each class as it does so, and stores them in an array of pairs (class id, distance) in the class structure. This distance metric used is the L1 Minkowski distance — it calculates the difference between each term, and sums that over all terms common to them[4]. Any characteristic terms not common to the classes will instead add a constant value to the distance. The value can be chosen by any means desired, and is definable in the header parameters. This will provide extra weight against picking classes with few terms in common. The term difference function used here is the same as the difference used in the comparisons in the standard weighted means method above (B.4).

Once it has these K nearest neighbors, the function will perform a standard

breadth first search over the classes, adding new ones to the search queue based on the neighbors of the current one. All classes reachable from a given starting class are set to be in the same cluster, defined by a simple integer id variable in the class. This search is attempted with every class as the starting point (as long as it wasn't already visited by a previous search). As a clustering operation, though, the one-way nature of the neighbor relation is undesireable. That is, simply because one class is the neighbor of a second does not mean the second class is a neighbor of the first. To deal with this, it also keeps track of a list of pairs (of cluster id numbers). If it happens upon a neighbor being part of an existing cluster, it will add a new pair to the list to be used in a later merging step. It will, of course, check for appearances of the current cluster in this list first, and if found it will instead add a pair which indicates a merge of the newly found cluster into that previous one, skipping the current cluster.

Once all classes have been visited, it will go through the merge list backwards, combining the values associated with the classes and changing the cluster id of any classes with the first cluster to use the second. Proceeding in this manner should avoid the issue of a previously removed cluster appearing later in the list. The search of this list performed when adding an entry will not, however, ensure that the cluster cannot appear again, as a constructed 'skip' pair could overlap with a previous entry. However, these entries should not alter the accuracy of the merge, they merely reduce the efficiency. Since there would have been a similar hit in weeding them out, and the merging step can be toggled in the header parameters, it was deemed acceptable to place the cost there. Finally, the action will push all classes onto a list in their associated cluster.

The classification step of this method is nearly identical to the standard weighted means approach. The difference comes after it has found the closest class. It will search through the cluster associated with this class, and rank the highest weighted differences between them and the document using the same formula as before. It will then apply these as secondary classes to the document, up to a number definable in the header parameters.

2.2.3 Support Vector Machines

The third classification method is a significant departure from the previous ones. It uses Support Vector Machines to create a decision function that is then applied to incoming documents. The implementation of the underlying SVM mechanics is done through the dlib library written by Davis King and provided under the Boost Software open source license[7]. The specific SVM implementation used is the Pegasos algorithm it contains, based on work by Shalev-Shwartz et al.[8]. This was chosen due to three key factors of the algorithm — it is online, converges quickly on large datasets, and can work with the sparse sample vectors as used by dlib[7]. Each of these is highly desirable traits for this application, allowing it to fit in nearly seamlessly with the operation of the full system. In particular, the sparse vectors are a natural fit to the characteristic term approach, as the majority of the features (the term information) are null for any given class and not stored by the classes themselves. Additional parameters used by the Pegasos implementation, as well as the kernel used, can be set in the header file, and the results from the values we used are provided in the results section.

However, the implementation provided is only the core binary classification, and thus the corresponding function versions for this method extend that to allow separation into many classes. The clustering action (which doesn't actually cluster for this instance), handles training the SVM decision functions. This will actually create a decision function for every class, which performs a binary classification with the intent of determining whether the incoming document is more likely part of this class or part of everything else (one versus all)[3]. To that end, it creates a sparse vector (map) out of the characteristic terms of each class, using a similar scoring function as seen above in order to reduce the term information into a single number (see B.2). Then it calls the dlib train function with this sample vector and either a 1 (for the class this function will be associated with) or a -1 (for every other class) as the label. Once this is finished, the resulting decision function is stored as part of the information of the class structure.

In order to use these, however, the classification action must first determine the characteristic terms of the document, something the above algorithms did not

require. This is performed in an auxiliary function (as it will be used again later), but runs in much the same manner as the computation for classes. The key differences being, of course, that the terms use document information rather than class, and the result is then stored as part of the document object. Once these are determined, the new sample will be fed into the decision function for each class, returning a value rating the class. The specifics of the value change based on the chosen kernel, but in all cases positive values reflect choosing the related class, and negative values should be the rest of the classes. Thus, the system will track the highest few results, with the specific number defined as a header parameter, and will return these as the primary and secondary classes of the document. If none of them are above zero, however, a new class is needed. Choosing such a class, however, is beyond the scope of the work here, and discussed in the future work section.

2.2.4 Jaccard Coefficients

The last classification method implemented is another fairly simple approach. At the core, Jaccard Coefficients are simply another way of calculating the difference between two sets of attributes, and thus there is no need for the clustering step in this algorithm. It actually uses an extended form of the Jaccard Coefficient meant for non-binary attributes, also known as the Tanimoto Coefficient[4].

Like the SVM algorithm, classification starts off by determining the characteristic terms of the new document. This will be used as the attribute vector of the document, to be compared against the respective vector of the classes. However, it first determines the squared magnitude of the vector, which is simply the sum of the term scores (using the same methods as before) squared. Then, for each class, it will determine the squared magnitude of its characteristic terms (the same way) and compute the dot product of the two vectors in parallel. To compute the dot product, the system will iterate through one set of characteristic terms, and attempt to find each term in the other set. If it exists, the product of the two term scores is added to the total. If not, the score for that term in the opposing set is treated as 0, and thus 0 will be added. This will ensure that two vectors which share no terms in common would have a dot product of 0, and thus be orthogonal under the

geometric interpretation. This would also result in a Jaccard Index of 0, the lowest possible value.

Once the dot product and magnitudes have been computed, the extended coefficient is calculated by dividing the dot product by the sum of the squared magnitudes of the two vectors minus the dot product (B.5). This is the score of the relationship between the document and this class, and the system will again keep track of the highest values it finds, as bounded by the header parameters. These will be the primary and secondary classes assigned to the document.

3. RESULTS AND FINDINGS

3.1 Performance Metrics

All of our testing was done on a database of books provided by Google Books. This database contains the text of one thousand books from the eighteenth and nineteenth centuries, though a few are from the late seventeenth century. We looked at two different categories of classification on these books, their publication date and subject. See Tables 3.1 and 3.2 for the distribution of the data set over these categories. Date was chosen because the books already contained the publication date in the meta data included at the top of the files. This made it easy to obtain the correct class to determine the accuracy with. For this we used two performance standards, the error and percent accuracy. The error is simply the average numeric difference between the assigned class and the class it should have had. The accuracy percentage is the percentage of times it was right, computed by keeping track of the number of misses.

Since some of the algorithms produce multiple secondary classes in descending strength, this is also applied to the miss rate. Specifically, the value added to the number of misses will be a function of where the correct class falls in the array of assigned classes. This number is 1 divided by the number of classes minus the index it appears at plus 1 (B.6). If it does not appear in the index will be regarded as equal to the number of classes, and thus the value will be 1. This means that the hit against the accuracy for a given placement will decrease according to the number of classes it can assign. As a result, the size of this array can have a significant impact on the returned accuracy beyond just the likelihood of a correct match ending up in the array. It is important to note, however, that this formula is only used for secondary classes. In other words, if the primary class (index 0) matches, nothing is added to the miss rate.

As mentioned, date is a very simple category to use. The downside of it, however, lies in the sample data. All of the documents are from a fairly narrow time period — the total spread is from 1675 to 1863, a little under two hundred years, in

Table 3.1: Date Distribution of Documents

Date	Count	Date	Count	Date	Count	Date	Count
1675	1	1684	1	1695	1	1698	1
1708	1	1715	1	1726	1	1729	1
1730	1	1732	1	1735	1	1736	1
1739	1	1741	1	1742	2	1749	1
1752	1	1757	1	1761	1	1762	1
1764	2	1766	1	1767	5	1768	1
1770	1	1771	2	1772	1	1773	3
1776	3	1777	2	1778	2	1779	4
1780	2	1781	1	1782	1	1783	1
1784	3	1786	2	1788	4	1789	2
1790	3	1792	3	1793	2	1794	3
1795	2	1797	2	1798	2	1801	5
1802	2	1803	3	1804	4	1805	7
1806	2	1807	4	1809	1	1810	1
1811	7	1812	5	1813	4	1814	6
1815	2	1816	7	1817	3	1818	1
1819	3	1820	5	1821	11	1822	12
1823	11	1824	14	1825	10	1826	15
1827	19	1828	16	1829	28	1830	16
1831	24	1832	18	1833	25	1834	20
1835	11	1836	15	1837	22	1838	16
1839	17	1840	13	1841	18	1842	24
1843	14	1844	24	1845	24	1846	16
1847	22	1848	20	1849	15	1850	20
1851	30	1852	17	1853	28	1854	31
1855	24	1856	25	1857	24	1858	26
1859	26	1860	23	1861	23	1862	24
1863	21						

109 classes. As you can see from the distribution in Table 3.1, however, 847 of them are in the 43 classes from 1821 to 1863 (43 years). Thus, there are many very close dates that will be likely to have been missed in the training set, and any differences between them should be relatively minor from a statistical perspective. There are also many classes with extremely few documents in them in the earlier time periods. This, as will be shown, tends to result in a fairly low average error, but a very high miss rate for classification based on date.

Subject, on the other hand, is the more useful category from a practical stand-

Table 3.2: Subject Distribution of Documents

Subject	Count	Subject	Count
Art	6	Biographical	84
Biology	11	Documentary	50
French	42	German	1
Historical	103	Italian	41
Latin	1	Literature	166
Mathematics	6	Medicine	42
Military	12	Philosophy	28
Political	73	Religion	139
Science	13	Social	26
Spanish	140	Sports	2
Technology	14		

point. Since the attribute is not numeric, the error rate is not applicable. As a result, the miss percentage is the only metric we used that is able to provide an idea of the accuracy. It also means that determining the correct class to compare it against is much trickier. As there was no supplied meta data for this, we ended up manually choosing classes and inserting them into the meta data of the documents. However, human error in trying to efficiently categorize nearly one thousand books fairly quickly could have a result on the reported accuracy. It is also important to note that, in order to reduce the effects of the extremely small classes as noted in the date distribution above, we erred on the side of more generic class names. As you can see in Table 3.2, we integrated some of the lesser used classes, for example "Physics" and "Chemistry" into a generic "Science", but left others like "Biology" alone. This was done to try and avoid having many singleton classes, but to still allow some similar subject types to see how clustering and secondary classes play into the results. Non-English texts were grouped by language for similar reasons.

The last important impact on the metrics is the training set versus classification set aspect of the system. Since the system as it stands is unable to invent new classes for documents, the ones that show up in the training set are the only ones that can be used (though an "other" category does exist in some of the methods). A poorly chosen training set, therefore, can cripple the results of the program. To try and mitigate this somewhat, we ran our results on four different training sets.

The first is simply the first fifty documents. The second was a fifty document chunk out of the middle of the set. The third was chosen with some focus to contain the major subject types. The last was an extension of the third to include twice as many documents in the training set. The second set was quickly discarded from most of the tests as it proved to provide extremely poor results as will be shown later. The fourth was only rarely used as a comparison of the effect of increasing the training set rather than the performance of the methods. See Tables A.1 through A.4 for the class distributions in the training sets.

3.2 Algorithm Performance

3.2.1 General Parameters

The globally used parameters defined in the header file are related to the general operations of the program. There is also the rate parameter specified as an argument. With a couple exceptions for comparison, we used rate = 16 on all of our test cases to achieve a decent speed. The first header define is a simple selection parameter to determine the method used. As mentioned in the metrics section above, the classTypes parameter specifies the number of classes that can be assigned to a document. This is specifically meant to be a way to get additional, similar classes to the one that was assigned. In other words, a document may straddle two different subjects, and this way they could both be provided. Increasing this also improves the likelihood that the correct class will be included in the assignment, and thus improve (if only moderately) the accuracy rating. Making it too high, however, dilutes the usefulness of categorization, as it could begin to fill in classes with a poor result just to fill in the array. Some of the methods do, however, have thresholds to weed out completely irrelevant classes from the result. We set classTypes to 3 in all of our featured test cases.

The topK parameter is similar in that it indicates the maximum number of characteristic terms assigned to classes and documents. As with the secondary classes, having more terms does not always improve the results, and can greatly impact the run time and memory required. As discussed in previous chapters, the characteristic terms are a heuristic used to reduce the number of comparisons

between objects. Lower numbers of terms are more likely to result in poor performance, as the number of ways they are being compared, and thus the potential differences between objects becomes smaller. However, having a high value means many comparisons and significantly more computations, depending on the method.

The term relevancy cutoff is determined by three parameters in the header. The first two, minKeep and supportScale, determine the cutoff for the term frequency. Essentially, the term must appear at a rate of at least minKeep times in supportScale documents to remain relevant. Anything below that is regarded as too infrequent to be useful. The last, minIDF, means that the term must have an idf above this number to be kept. Again, if it is lower than this item it is regarded as appearing in too many different documents to have any distinguishing power. This formula is given in B.3. This can have a significant impact on the results as well, in the same way that the number of characteristic terms affects the results. If these are too high and cut out too many terms, then there will be little to differentiate the objects by. On the other hand, making it too low will result in slower performance and bloated knowledge files and memory requirements. It is also worth noting that the issues mentioned below with the scoring function may also apply here, as it is a similar normalization by document count formula. However, we did not test scaling by term counts instead.

The last global parameter, scalar, relates to the term scoring function. Although it does not necessarily provide much as far as a term ranking is concerned (specifically, the characteristic term computation), it can be very helpful in clustering. Essentially, it can scale the term scores up, and this can mean larger differences between the scores of terms. As a result, it can provide more space between classes and allow for easier categorization. This is not very useful in the classification step due to the nature of the functions used. They are all based on the relative ranking of the terms, which will not be changed by a scalar multiple. However, clustering depends on the relative distance between objects, and thus the scaling can have an impact there. If it is too large, though, it can space things out too much and push everything above the method thresholds.

As briefly hinted at in the previous chapter, the scoring function itself can be

considered a global parameter, although it is defined in the algorithms file rather than the header. The original formula is given in B.2, and is a fairly standard value relating the term frequency to the idf. However, it computes the frequency by normalizing the count by the number of documents it appears in. This could potentially have trouble on documents of extremely small or large sizes. Such documents would have disproportionately few or very many occurrences of terms when compared to the mean, and as a result the score could be adversely affected. To look into this, we also used a version of the function that normalized by the total number of terms at the associated level (document, class, global), as shown in B.7. Another potential modification is to take the logarithm of the term frequency in place of, or in addition to, the normalization scaling[3]. This was not part of our test cases, however.

3.2.2 Jaccard Coefficients

As the Jaccard Coefficients method does not involve a clustering step or any parameters specific to its method, it can be a good way of testing the effects of the global parameters. We tested rather extensively over these parameters, and ultimately determined that the individual values are less important that the relations between them. The balance is dependent on the relation between the number of characteristic terms (defined by topK) and the number of global terms (defined by the interplay between minKeep, supportScale, and minIDF). Scalar ultimately seemed to be irrelevant, mostly likely due to the lack of clustering.

As you can see in the error and accuracy results in the following tables, a small number of characteristic terms results in terrible performance. This can result either from a small value of topK, or from a small number of global terms. Increasing topK will improve results, to a certain point. However, there appears to be a threshold beyond which there is not noticeable impact beyond an increasing run time. As such, most of our test cases stuck with topK = 250. This produced reasonable performance (though not the best recorded) at about one hour per case (training and testing combined).

Thus, this suggests the method of determining global parameters. Increasing topK will improve the accuracy to a certain threshold. The point at which there

Table 3.3: Jaccard Coefficients results on date category

Parameter	Set	Error	Accuracy	Additional
scale = 100	1	76.3179	4.75%	< 250 terms left
scale = 100	1	973.994	2.30%	rate = 4, most in "other"
scale = 250	1	26.2737	3.79%	
scale = 300	1	21.7053	5.11%	
scale = 300	1	21.7053	5.11%	scalar = 250
scale = 300	1	21.7053	5.11%	scalar = 750
scale = 300	1	37.2558	3.32%	minIDF = 0.05
scale = 300	1	71.4737	3.32%	minIDF = 0.25
scale = 325	1	22.7926	4.61%	
scale = 400	1	22.1842	4.19%	
scale = 500	1	22.3737	3.70%	
scale = 300	2	731.593	3.37%	Put most in "other"
scale = 250	3	20.6695	3.49%	
scale = 300	3	21.5316	3.72%	
scale = 325	3	20.5011	3.65%	
scale = 500	3	23.0779	3.44%	
scale = 300	4	18.9422	4.19%	

ceases to be a benefit is determined first by the relevancy cutoffs. If there are too few terms to make adequate use of the limit, topK will effectively be set to the total number of terms. This will mean less effectiveness. Lowering the cutoffs (increasing supportScale, decreasing minKeep or minIDF) will result in more terms to counter this. It is worth noting that this adequate number does not seem to simply be equal to topK (though in any case, it would be very difficult to set the cutoff to achieve an exact number of remaining terms), but rather several times more than it. After raising the number of total terms, the performance should increase, but will be capped again by a maximum result. This likely depends on the training set, data set, and method used. There is also the performance consideration in play, as increasing topK will result in longer runtimes. Some applications may desire an accuracy hit to get faster speed. Our test cases generally stuck with minKeep = 1000 and minIDF = 0.1, and varied supportScale to change the cutoff. We found that, given the above typical cases, supportScale $\geq$ 250 worked well. Increasing beyond that did not change the performance much, unless other parameters were also modified. See Tables 3.3 through 3.6 for results, and Tables A.5 through A.7

Table 3.4: JC results on date with by-document normalization

Parameter	Set	Error	Accuracy	Additional
scale = 100	1	1588.06	1.37%	most in "other", < 250 terms
scale = 200	1	1436.58	2.84%	most in "other"
scale = 300	1	27.4811	4.63%	
scale = 400	1	37.5126	4.65%	
scale = 500	1	43.9389	3.81%	
scale = 250	3	17.8147	4.12%	
scale = 300	3	19.3221	4.89%	
scale = 325	3	19.3621	4.98%	
scale = 500	3	19.7137	5.05%	

Table 3.5: Jaccard Coefficients results on subject category

Parameter	Set	Accuracy	Additional
scale = 250	1	22.0526%	
scale = 300	1	23.5965%	
scale = 325	1	21.4737%	
scale = 500	1	23.1930%	
scale = 300	2	21.4900%	many in "other"
scale = 250	3	39.5263%	
scale = 250	3	27.1754%	rate = 4
scale = 300	3	31.7018%	
scale = 325	3	31.0000%	
scale = 500	3	33.0175%	
scale = 500	3	42.2105%	topK = 500
scale = 750	3	39.7368%	topK = 750
scale = 250	4	42.4074%	

Table 3.6: JC results on subject with by-document normalization

Parameter	Set	Accuracy	Additional
scale = 250	1	27.7719%	
scale = 300	1	32.0351%	
scale = 325	1	32.2281%	
scale = 500	1	26.1404%	
scale = 500	1	17.2281%	topK = 500
scale = 250	3	27.7719%	
scale = 300	3	32.0351%	
scale = 325	3	32.2281%	
scale = 500	3	26.1404%	
scale = 500	3	40.9123%	topK = 500

for selected result distributions.

3.2.3 Weighted Means and Clustering

Given how similar these are, the results will be discussed in tandem. First, however, the clustering algorithm has some additional header parameters associated with its operation. The nearK parameter sets the max number of neighbors that will be chosen for each class. Higher values can result in blurry boundaries between clusters (and thus potentially merging clusters that would otherwise have been separate). However, small values can result in more isolated and smaller clusters, if there are clique-like formations, which may also be less useful.

The second, mergeClusters, is a binary parameter that toggles whether or not clusters should be merged if members have neighbors across clusters. This is mainly to account for the one-way aspect of the neighbor relation. If two classes do not have each other as mutual neighbors, then it is possible the search will miss one (and potentially many others connected to it) even if that class had the first as a neighbor. In some cases, however, this may be a desired way to try and isolate these one-way relations, and thus the option is provided to disable merging. All of our test cases left merging enabled.

The third is maxD, a definition of the maximum distance to be considered a neighbor. In other words, even if a given class is the closest one to the currently examined one, if that distance is greater than maxD it will not be included as a neighbor. This, therefore, can be used to help define the spread between clusters. Setting it low will result in more isolated and smaller clusters, while setting it high will result in larger clusters. This is particularly important with respect to the scalar parameter. Since maxD defines the spread threshold, and scalar will multiply the standard results (increasing or decreasing the spread), the two share a relationship similar to the one between topK and the cutoffs discussed above.

The last is penalty, which is the constant value added to the distance for each term not common to the compared classes. In other words, if one of them has a characteristic term the other does not, it will add this value in place of being able to calculate a term distance. This is again heavily tied to scalar and maxD, as high

Table 3.7: Weighted Means results on date

Parameter	Set	Error	Accuracy	Additional
scale = 100	1	109.968	0%	< 250 terms
scale = 300	1	108.535	0%	
scale = 500	1	106.768	0%	
scale = 100	3	109.968	0%	< 250 terms
scale = 300	3	108.535	0%	
scale = 500	3	106.768	0%	

values of this will increase the distances between classes. We left it as 1000 in all of our tests.

Weighted means is unaffected by scalar, given the lack of a clustering action. Thus, like Jaccard Coefficients, the scoring function and cutoff parameters are the most important factors in performance. As you can see from Tables 3.7 to 3.10, the results of this are very poor. The second scoring function, B.7, produced terrible results that are actually worse than a random selection. From the distributions, one of which is provided in Table A.8, it looks as though it simply shoved everything into the first class, suggesting that the scoring function was simply ineffective over all classes and regardless of the parameters tested. The original function, B.2, produced much better results, although it is still below the values provided by Jaccard Coefficients. In particular, the accuracy results are far worse. While the better returns gave comparable error in the date category, the more useful metric there, this cannot be avoided with regards to the subject category. This is likely because the standard weighted means does not provide secondary classes, and thus it has no "second chances" to get the correct class. The cutoff parameters also seem to follow a similar pattern as the Jaccard Coefficients, in that more terms results in better performance, to a certain point. Beyond this, and they dilute the search space and instead cause more harm. As a result, this method is not very effective in classifying subjects, and provides little value over Jaccard Coefficients with regards to date. Our included test cases all used minKeep = 1000, minIDF = 0.1, and scalar = 500, and varied supportScale.

The clustering method is very sensitive to its parameters. If it is unable to

Table 3.8: WM results on date with by-document normalization

Parameter	Set	Error	Accuracy	Additional
scale = 100	1	30.7126	2.42%	< 250 terms
scale = 300	1	25.3568	1.05%	
scale = 500	1	22.3747	1.47%	
scale = 100	3	33.5453	1.79%	< 250 terms
scale = 300	3	31.3611	1.16%	
scale = 500	3	19.3242	1.68%	

Table 3.9: Weighted Means results on subject

Parameter	Set	Accuracy	Additional
scale = 100	1	0.526316%	< 250 terms
scale = 300	1	0.526316%	
scale = 500	1	0.526316%	
scale = 100	3	0.526316%	< 250 terms
scale = 300	3	0.526316%	
scale = 500	3	0.526316%	

cluster classes together, performance is no better than standard weighted means, as there will be no secondary classes and the classification scheme is identical. So the values must be set to keep them close enough together to allow clustering to occur. This is mainly through the interplay between maxD and scalar. Increasing scalar will increase the distance between terms, and thus classes. If it is too high, then more classes will exceed maxD and be excluded from the potential neighbors.

Unlike other methods, topK can have a large impact here as well. Since terms that are uncommon to the characteristic sets will count against the distance, a large

Table 3.10: WM results on subject with by-document normalization

Parameter	Set	Accuracy	Additional
scale = 100	1	4.31579%	< 250 terms
scale = 300	1	7.36842%	
scale = 500	1	3.68421%	
scale = 100	3	4.42105%	< 250 terms
scale = 300	3	11.4737%	
scale = 500	3	9.15789%	

Table 3.11: Clustering results on date with by-document normalization

Parameter	Set	Error	Accuracy	Additional
scale = 100	1	30.7126	2.75%	topK = 250 par. set, < 250 terms
scale = 100	1	20.8768	2.63%	topK = 500 par. set, < 500 terms
scale = 300	1	25.3568	2.44%	topK = 250 par. set
scale = 300	1	23.7063	3.53%	topK = 500 par. set
scale = 500	1	22.3747	3.12%	topK = 250 par. set
scale = 500	1	20.7547	4.33%	topK = 500 par. set
scale = 100	3	33.5353	2.25%	topK = 250 par. set, < 250 terms
scale = 100	3	26.1705	2.98%	topK = 500 par. set, < 500 terms
scale = 300	3	31.3611	2.61%	topK = 250 par. set
scale = 300	3	18.4326	3.95%	topK = 500 par. set
scale = 500	3	19.3242	3.79%	topK = 250 par. set
scale = 500	3	30.9905	3.88%	topK = 500 par. set

topK will mean higher distances between classes. Thus, if trying to improve accuracy with more characteristic terms, this must be accounted for when choosing the other parameters. Specifically, increasing maxD will raise the cutoff for neighbors, and decreasing scalar will make the values provided by the common terms smaller. Both of these will help, and are best used in conjunction. Decreasing penalty could also counteract this, though we did not explicitly test for that. However, changing them beyond a certain point will only cause more problems, meaning all parameters must be tweaked and tested for the optimal balance.

This makes clustering more difficult to set up properly than some of the other methods. Our test cases all used minKeep = 1000 and minIDF = 0.1 like the previous tests, and penalty = 1000 as noted above. We also used, for run time constraints, topK = 250 and topK = 500. Testing showed that we received the best results for the first value with scalar = 200 and maxD = 250,000. For 500, these were scalar = 100 and maxD = 500,000. Comparatively, however, the run time was generally the quickest of the three more complicated algorithms, for the same values of topK and cutoffs, and it produced good error results in the date category. Its subject accuracy did not reach the same levels, however. See Tables 3.11 and 3.12 for the results, and Tables A.9 and A.10 for some sample result distributions.

Table 3.12: Clustering results on subject with by-doc. norm.

Parameter	Set	Accuracy	Additional
scale = 100	1	5.89474%	topK = 250 par. set, < 250 terms
scale = 100	1	10.8246%	topK = 500 par. set, < 500 terms
scale = 300	1	13.1930%	topK = 250 par. set
scale = 300	1	13.8421%	topK = 500 par. set
scale = 500	1	14.7544%	topK = 250 par. set
scale = 500	1	8.66667%	topK = 250, scalar = 250
scale = 500	1	6.47368%	topK = 250, scalar = 300
scale = 500	1	3.68421%	topK = 250, scalar = 1000
scale = 500	1	19.2281%	topK = 500 par. set
scale = 500	1	4.31579%	topK = 500, scalar = 150, max = 250,000
scale = 100	3	7.01754%	topK = 250 par. set, < 250 terms
scale = 100	3	10.1053%	topK = 500 par. set, < 500 terms
scale = 300	3	18.3333%	topK = 250 par. set
scale = 300	3	19.7719%	topK = 500 par. set
scale = 500	3	17.4035%	topK = 250 par. set
scale = 500	3	6.12281%	topK = 500 par. set

3.2.4 Support Vector Machines

The dlib implementation of pegasos provides several parameters of its own, the most relevant of which are included in the header defines. As explained in the dlib documentation[7], the first of these is lambda, which represents the tradeoff of exact fitting versus generalization. In other words, having a small value of lambda will encourage the trainer to fit the classes exactly, while larger values will allow some errors to get more generalized results. The default value of this is 0.0001.

The other two parameters are tol, the tolerance, and maxVect, the maximum number of support vectors allowed. These two are related, in that tolerance will determine the number of support vectors used to represent the decision function, but cannot exceed maxVect. Smaller values of tolerance use more vectors, and thus should more accurately capture the criteria. The default values of these are 0.01 and 40 respectively.

Lastly, while not a parameter per se, the kernel type is also defined in the header. The dlib library provides four kernels that work with sparse vectors, three of which can be used in our program. These kernels are polynomial, radial_basis,

Table 3.13: Support Vector Machines results on date

Parameter	Set	Error	Accuracy	Additional
scale = 100	1	35.3232	3.39%	< 250 terms
scale = 300	1	24.5053	3.00%	
scale = 300	1	23.8621	3.32%	lambda=0.001
scale = 300	1	32.2558	3.74%	lambda=0.001, tol=0.1
scale = 300	1	37.9632	4.75%	lambda=0.001, vect=20
scale = 300	1	28.5432	3.05%	lambda=0.001, vect=20, scalar=250
scale = 300	1	60.5389	2.47%	lambda=0.001, vect=20, scalar=1000
scale = 300	1	33.0400	2.61%	topK=100, lambda=0.001
scale = 300	1	88.6295	2.72%	topK=500, lambda=0.001
scale = 300	1	53.4221	4.09%	lambda=0.01
scale = 300	1	35.6326	4.47%	lambda=0.01, vect=20
scale = 300	1	34.1832	4.39%	lambda=0.01, vect=20, tol=0.1
scale = 300	1	69.6747	2.98%	lambda=0.00001
scale = 300	1	24.3000	2.98%	tol=0.1
scale = 300	1	25.9379	3.28%	tol=0.001
scale = 300	1	25.6284	3.28%	vect=20
scale = 300	1	24.5053	3.00%	vect=75
scale = 500	1	33.8611	2.60%	
scale = 500	1	23.6684	5.63%	topK=500, lambda=0.001
scale = 100	3	29.9684	3.31%	< 250 terms
scale = 300	3	25.3011	3.84%	
scale = 500	3	24.3063	2.86%	

and sigmoid. Our preliminary testing suggested that sigmoid works the best, so that is the default kernel.

As you can see from the results in Tables 3.13 through 3.16, SVM is very hard to set up. The parameters all play into each other and interact behind the scenes to affect the results. With one lambda, increasing the tolerance or number of characteristic terms may produce much better results. With another, it might completely destroy them. This is most likely due to the nature of the margin classification scheme used in SVM. A small lambda means the the algorithm will try and fit more exactly to the data. This would intuitively suggest more accuracy, but may fall prey to over fitting. Meaning, it tries so hard to match the data that it can't deal with things that aren't exactly the same as what it's seen before. Tolerance, and the related maximum number of vectors, can have a similar impact. Low

Table 3.14: SVM results on date with by-document normalization

Parameter	Set	Error	Accuracy	Additional
scale = 100	1	23.3926	4.21%	< 250 terms
scale = 300	1	23.7937	4.11%	
scale = 500	1	24.2326	3.49%	
scale = 100	3	23.3137	5.35%	< 250 terms
scale = 300	3	23.8874	4.82%	
scale = 500	3	23.4674	4.11%	

Table 3.15: Support Vector Machines results on subject

Parameter	Set	Accuracy	Additional
scale = 100	1	2.94737%	< 250 terms
scale = 300	1	10.7018%	
scale = 300	1	7.89474%	lambda=0.1
scale = 300	1	10.8947%	lambda=0.1, tol=0.1
scale = 300	1	21.7193%	lambda=0.01
scale = 300	1	21.7193%	lambda=0.01, vect=20
scale = 300	1	4.94737%	lambda=0.01, tol=0.1
scale = 300	1	5.75439%	lambda=0.01, topK=500
scale = 300	1	6.87719%	lambda=0.001
scale = 300	1	6.87719%	lambda=0.001, vect=20
scale = 300	1	7.26316%	lambda=0.001, tol=0.1
scale = 300	1	1.85965%	lambda=0.00001
scale = 500	1	7.63158%	
scale = 100	3	7.85965%	< 250 terms
scale = 300	3	7.15789%	
scale = 300	3	3.03509%	lambda=0.1
scale = 300	3	18.6667%	lambda=0.1, tol=0.1
scale = 300	3	9.91228%	lambda=0.01
scale = 300	3	4.36840%	lambda=0.01, tol=0.1
scale = 500	3	3.36842%	

Table 3.16: SVM results on subject with by-document normalization

Parameter	Set	Accuracy	Additional
scale = 100	1	10.9298%	< 250 terms
scale = 300	1	7.66667%	
scale = 500	1	4.19298%	
scale = 100	3	10.9123%	< 250 terms
scale = 300	3	7.84211%	
scale = 500	3	2.91228%	

values of tolerance will use more vectors to represent the decision function, which means a more exact representation of it. Again, this can lead to over fitting and make the function unable to generalize to new documents. On the other hand, of course, making these values too large will undermine the whole action it is trying to accomplish — it won't be separating documents with any attempt at matching the data it has. As a result, much like clustering showed, there is a "sweet spot" in the interaction of these parameters.

This is further compounded when our program parameters are added in. The number and quality of the characteristic terms can, if not chosen properly, provide more ways for the algorithm to over fit or too little to be useful. Again, the best selections seem to be dependent on the other parameter values, making it even more difficult to achieve the best results. Even the most accurate results we were able to find with SVM, however, did not match up to the ones received from Jaccard Coefficients. There is potential for it to perform well, but unless the parameters can be chosen effectively the usefulness of this method is questionable. Selected result distributions are available in Tables A.11 and A.12.

4. CONCLUSIONS AND FUTURE WORK

As is evident from the tables in the previous chapter, the results are not inspiring. None of our measures, under any parameter set we could find, achieved even 50% accuracy for subject categorization. With date as the class, accuracy was rarely above 5%, with the lowest error being just under 19 years. These results do show a significant improvement over a random guess — with 109 dates and 21 subjects a purely random selection would be expected to average about 0.917% and 4.76% respectively. Factoring in the secondary classes we allowed, these numbers would increase to about 2.75% and 14.29%, still much lower than our best results. However, from a practical standpoint, getting it "correct" less than half the time is not desirable. Even discarding the accuracy for the date category, since the nature of the class makes exact matches difficult, does not help. The minimum average error is not actually as good as it might seem, given that 84.7% of the documents fall in a 43 year range. Indeed, looking at the distributions of the selected results in Tables A.5 through A.12, the methods usually placed the majority of the documents into one or two classes in this range, rather than demonstrating an even spread, or one representational of the distribution in the corpus. All of this suggests that, while the methods did provide a better-than-chance categorization, they are not very useful in a practical sense.

The question then becomes why this is the case, and if anything can be done to improve this showing. One immediate answer is that our parameter selection could be poor. All of the schemes, though especially so for clustering and SVM, are very sensitive to their parameters, and the interaction between them makes it difficult to simply pick the best from each individual value and put them together. Cross-validation techniques could be used to help select parameters, though they may not be practical on very large data sets like our intended field. It may be worth exploring more intelligent methods of choosing parameters in the future.

Another possibility is that our chosen methods are algorithmically stupid. They are naive in that they perform only statistical analysis — there is no attempt at

learning or using the semantic meanings of terms or the syntactical relations between them. It does not even use n-gram groupings of terms to compute frequencies over[1]. There have been many forays into these more advanced techniques, such as the one by Ge and Mooney, and it is quite possible that they would prove to be more effective on our data sets[9].

However, it is important to recall the reasons we chose statistical classification in the first place. As noted in Chapter 1, statistical methods have shown to be more readily scalable to extremely large data sets and less affected by broken language in the documents[1]. A quick look at the text in our documents and the terms the program is reading in will show that this is critical. Misspellings and broken grammar are prevalent in large numbers, and special characters are dotted throughout (even in the middle of words). This is a typical issue with OCR data pulled from old books, and ones with stylistic fonts, such as those in our data set[2]. All of this suggests that semantic and syntactic analysis would have a hard time compensating for the peculiarities of the corpus.

Is it, then, a task beyond the current computational methods? Halevy et al. again suggest not[1]. While the corpus they are referring to, the web, is very different from our intended field of use, scanned documents, the core point remains the same. This is further demonstrated by Hays and Efros in yet another field, images[10]. It is the size of the available data that is the most important factor in statistical classification. By accumulating enough data, the unusual terms fade out and the simple frequency analysis can shine.

However, the size of our set is still limited, and in most cases it is not practical to assume that manual classification of a significant portion of the documents will be possible. Indeed, this was one of the reasons we chose semi-supervised learning methods in the first place. As such, our results show that the personal use of statistical classification methods do not work well for books. While it is possible that certain highly discriminatory special cases may exist (such as sorting two kinds of documents with no terms common to them), the general use of these methods seems tied to the corpus. In that case, however, it may still prove to be useful in our intended field. If a large library's worth of document text is used as the

training set, then our statistical methods may prove to be much more accurate and practical. Snow et al. also discuss the viability of using public annotations, which could potentially be useful in creating a larger, labeled training set[11]. We were unable to obtain such a sizable corpus, however, so this must be left for future researchers.

Even with that taken care of, there is still room for our implementations to be extended further. Most notably, the program can separate out documents that it can't find a good class for into an "other" category. Attempting to generate a new class label for these documents may be possible, but it would require extending our system, and would also need to be based on semantic and syntactic rules. In some instances, however, it might be desirable for the program to pause and solicit a class from a human operator. This would allow the new class (or adjusted old class) to be applied to the remainder of the results, but would require extending our work to allow online supervision.

Another potentially useful extension would be to make more practical use of the secondary classes. Currently, they are only used in calculating the accuracy metric, and the document is assigned to only the primary class. However, a probabilistic confidence measure like the one used by Subramanya and Bilmes could make good use of these to improve the accuracy of the assigned classes[12]. It may also be worth examining the possibility of modifying the methods to truly sort documents into multiple classes, as it is certainly possible for one to straddle multiple subjects or genres. This would provide a potentially very valuable resource when compared against common book classification schemes like those employed by libraries. These schemes pigeonhole every document into only one class, but allowing multiple classes would enhance searching. Adjustments to the method implementations which factor these in could also be worth exploring.

Given all of this, there is still a great deal of potential in statistical classification in this application. However, without an extremely large corpus to train against, more intelligent methods will remain the more effective option.

LITERATURE CITED

[1] Alon Halevy, Peter Norvig and Fernando Pereira. "The Unreasonable Effectiveness of Data". *IEEE Intelligent Systems*, volume 24 Issue 2, pages 8–12. 2009.

[2] Luc Vincent. "Google Book Search: Document Understanding on a Massive Scale". *International Conference on Document Analysis and Recognition*. Curitiba, Brazil. Sept. 2007.

[3] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze. *An Introduction to Information Retrieval*. Cambridge University Press. 2008.

[4] Pang-Ning Tan, Michael Steinbach and Vipin Kumar. *Introduction to Data Mining*. San Franciso: Pearson Addison Wesley. 2006.

[5] Heng Ji and Ralph Grishman. "Data Selection in Semi-supervised Learning for Name Tagging". *Workshop on Information Extraction Beyond the Document*, Association for Computation Linguistics, pages 48–55. July 2006.

[6] Jingyang Li and Maosong Sun. "Scalable Term Selection for Text Categorization". *Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, Association for Computational Linguistics, pages 774–782. June 2007.

[7] Davis E. King. dlib C++ Library. http://dclib.sourceforge.net/. Last accessed July 2, 2009.

[8] Shai Shalev-Schwartz, Yoram Singer and Nathan Srebro. "Pegasos: Primal estimated sub-gradient solver for SVM". *International Conference on Machine Learning*. Corvallis, OR. 2007.

[9] Ruifang Ge and Raymond J. Mooney. "A Statistical Semantic Parser that Integrates Syntax and Semantics". *Computational Natural Language Learning*, Association for Computational Linguistics, pages 9–16. June 2005.

[10] James Hays and Alexei A. Efros. "Scene Completion Using Millions of Photographs". *ACM Transactions on Graphics*, volume 26, issue 3, article 4, 7 pages. 2007.

[11] Rion Snow, Brendan O'Connor, Daniel Jurafsky and Andrew Y. Ng. "Cheap and Fast — But is it Good? Evaluating Non-Expert Annotations for Natural Language Tasks". *Empirical Methods in Natural Language Processing*, pages 254–263. Oct. 2008.

[12] Amarnag Subramanya and Jeff Bilmes. "Soft-Supervised Learning for Text Classification". *Empirical Methods in Natural Language Processing*, pages 1090–1099. Oct. 2008.

APPENDIX A
Additional Tables Referenced

Table A.1: Distributions of Training Set 1

Subject	Count	Date	Count	Date	Count	Date	Count
Art	1	1726	1	1762	1	1767	1
Biographical	7	1807	1	1812	1	1816	1
Biology	2	1822	1	1827	3	1828	4
Documentary	8	1829	1	1830	2	1831	1
Historical	8	1833	1	1834	1	1836	1
Literature	4	1837	1	1838	2	1839	1
Medicine	6	1840	1	1843	4	1844	1
Military	2	1845	2	1846	2	1847	1
Philosophy	2	1848	1	1849	1	1850	2
Political	1	1851	1	1853	1	1855	1
Religion	3	1856	1	1858	4	1860	2
Social	5						
Sports	1						

Table A.2: Distributions of Training Set 2

Subject	Count	Date	Count	Date	Count	Date	Count
Biographical	14	1777	1	1794	1	1820	2
Documentary	4	1823	3	1824	1	1826	1
Historical	13	1828	1	1829	2	1837	2
Literature	7	1838	1	1841	2	1843	1
Medicine	1	1845	1	1847	1	1848	1
Philosophy	1	1850	1	1851	4	1852	2
Political	5	1853	2	1854	2	1855	1
Religion	5	1856	3	1857	2	1858	3
		1859	4	1860	1	1861	1
		1863	3				

Table A.3: Distributions of Training Set 3

Subject	Count	Date	Count	Date	Count		
Art	1	1726	1	1762	1	1767	1
Biographical	2	1807	2	1812	1	1815	1
Biology	2	1816	1	1822	1	1827	2
Documentary	7	1828	4	1829	2	1831	1
French	3	1832	1	1833	2	1834	1
Historical	9	1836	2	1837	1	1838	2
Italian	3	1839	2	1840	1	1843	3
Literature	3	1844	1	1845	1	1847	2
Medicine	4	1848	1	1849	1	1850	2
Military	1	1851	2	1854	2	1855	1
Philosophy	2	1858	1	1860	3		
Political	2						
Religion	3						
Social	4						
Spanish	3						
Technology	1						

Table A.4: Distributions of Training Set 4

Subject	Count	Date	Count	Date	Count	Date	Count
Art	1	1726	1	1762	1	1766	1
Biographical	5	1767	1	1772	1	1783	1
Biology	2	1784	1	1789	1	1790	1
Documentary	7	1792	1	1794	2	1805	1
French	4	1806	2	1807	2	1812	1
Historical	15	1815	1	1816	2	1817	1
Italian	4	1822	2	1823	1	1825	1
Literature	11	1827	2	1828	5	1829	3
Mathematics	2	1830	2	1831	2	1832	1
Medicine	8	1833	3	1834	1	1836	2
Military	1	1837	1	1838	3	1839	3
Philosophy	5	1840	1	1842	1	1843	3
Political	2	1844	2	1845	2	1846	1
Religion	15	1847	4	1848	1	1849	1
Science	6	1850	4	1851	4	1853	3
Social	5	1854	3	1855	1	1856	5
Spanish	3	1857	3	1858	1	1860	3
Technology	4	1861	1	1862	1		

Table A.5: Sample result distribution from set = 2 in Table 3.3

Date	Count	Date	Count	Date	Count	Date	Count
-1	373	1777	7	1794	4	1820	2
1823	20	1824	1	1826	1	1828	4
1829	25	1837	2	1838	4	1841	3
1843	1	1845	1	1847	1	1848	6
1850	32	1851	271	1853	2	1854	152
1855	57	1856	3	1857	12	1858	3
1859	4	1860	1	1861	1	1863	3

Table A.6: Sample result distribution from scale = 325 in Table 3.4

Date	Count	Date	Count	Date	Count	Date	Count
1726	1	1762	1	1767	1	1807	2
1812	1	1815	1	1816	2	1822	1
1827	2	1828	8	1829	392	1831	1
1832	1	1833	3	1834	1	1836	2
1837	1	1838	2	1839	33	1840	9
1843	3	1844	2	1845	1	1847	2
1848	513	1849	1	1850	2	1851	3
1854	2	1855	1	1858	1	1860	4

Table A.7: Sample result distribution from topK = 500 in Table 3.5

Subject	Count	Subject	Count	Subject	Count	Subject	Count
Art	1	Biographical	141	Biology	3	Documentary	8
French	41	Historical	43	Italian	39	Literature	634
Medicine	48	Military	1	Philosophy	2	Political	2
Religion	28	Social	4	Spanish	4	Technology	1

Table A.8: Sample result distribution from 300, 1 in Table 3.9

Subject	Count	Subject	Count	Subject	Count	Subject	Count
Art	951	Biographical	7	Biology	2	Documentary	8
Historical	8	Literature	4	Medicine	6	Military	2
Philosophy	2	Political	1	Religion	3	Social	5
Sports	1						

Table A.9: Sample result distribution from 300, 3, 500 in Table 3.11

Date	Count	Date	Count	Date	Count	Date	Count
1726	1	1762	1	1767	1	1807	36
1812	1	1815	1	1816	1	1822	1
1827	2	1828	4	1829	2	1831	1
1832	14	1833	24	1834	1	1836	2
1837	1	1838	2	1839	35	1840	530
1843	3	1844	1	1845	254	1847	28
1848	15	1849	1	1850	2	1851	2
1854	28	1855	1	1858	1	1860	3

Table A.10: Sample result distribution from 300, 3, 500 in Table 3.12

Subject	Count	Subject	Count	Subject	Count	Subject	Count
Art	1	Biographical	2	Biology	36	Documentary	7
French	410	Historical	9	Italian	100	Literature	3
Medicine	71	Military	1	Philosophy	2	Political	2
Religion	3	Social	4	Spanish	348	Technology	1

Table A.11: Sample result distribution from 500, 1, 500 in Table 3.13

Date	Count	Date	Count	Date	Count	Date	Count
-1	2	1726	1	1762	1	1767	1
1807	17	1812	2	1816	1	1822	1
1827	9	1828	5	1829	21	1830	69
1831	31	1833	77	1834	84	1836	1
1837	3	1838	2	1839	2	1840	6
1843	11	1844	16	1845	3	1846	6
1847	5	1848	9	1849	18	1850	7
1851	72	1853	207	1855	57	1856	12
1858	93	1860	148				

Table A.12: Sample result distribution from 300, 1, 0.01 in Table 3.15

Subject	Count	Subject	Count	Subject	Count	Subject	Count
Art	17	Biographical	8	Biology	29	Documentary	28
Historical	29	Literature	17	Medicine	10	Military	7
Philosophy	4	Political	25	Religion	467	Social	163
Sports	196						

APPENDIX B
Referenced Formulas

The inverse document frequency (idf) of a term is defined as:

$$log_{10}\frac{docCount}{termDocCount} \tag{B.1}$$

where docCount is the total number of documents and termDocCount is the number of documents containing the given term.

The original term score function:

$$scalar \times idf \times \frac{termCount}{termDocCount} \tag{B.2}$$

where termCount is the number of occurrences of the term at this level.

A term will be discarded (deemed irrelevant) if the following is true:

$$termIDF < minIDF \parallel \frac{termCount}{termDocCount} < minKeep \times \frac{docCount}{supportScale} \tag{B.3}$$

where minIDF, minKeep, and supportScale are constants defined in the header, and termIDF is the idf of the term.

Weighting function for the Weighted Means and Clustering methods:

$$\sum_{i \in C} scalar \times term_i IDF \times \left| \frac{term_i Count}{classDocCount} - term_i docCount \right| \tag{B.4}$$

where C is the set of characteristic terms in the class.

The Extended Jaccard Coefficient between two vectors p and q:

$$\frac{p \cdot q}{||p||^2 + ||q||^2 - p \cdot q} \tag{B.5}$$

The value added to a miss according to the index i where it appears:

$$\frac{1}{classTypes - i + 1} \tag{B.6}$$

The modified term score function, with term count normalization:

$$scalar \times idf \times \frac{termCount}{levelCount} \tag{B.7}$$

where levelCount is the number of terms at the level calling the function.

APPENDIX C
Supplemental Files

All of our code was developed using Xcode on Mac OS 10.4 and is written in C++. It was additionally tested and run on Windows XP using the MinGW g++ port. The supplied program requires the dlib library[7], though removal of all SVM-related functions and attributes would allow it to compile and run with the remaining three methods. The source code is included in an attached CD with the paper copy of this thesis, and is also available online. The CD contains the following files:

main.cpp.txt	10,357 bytes	The main program
algorithms.cpp.txt	21,472 bytes	The functions containing method specific implementations
algorithms.h.txt	946 bytes	The function declarations and includes for algorithms.cpp.txt
DocItem.h.txt	2,161 bytes	The class and structure definitions for objects
DocItem.cpp.txt	1,263 bytes	The implementations of functions defined in DocItem.h.txt
classifier.h.txt	1,187 bytes	The definitions of global parameters that tweak method operation