A Framework for Text Categorization

A thesis submitted in fulfillment of the requirements for the degree of

Master of Engineering (Research)

by

Ken Williams

School of Electrical and Information Engineering

The University of Sydney

March 18, 2003

Abstract

The field of automatic Text Categorization (TC) concerns the creation of categorizer functions, usually involving Machine Learning techniques, to assign labels from a pre-defined set of categories to documents based on the documents' content. Because of the many variations on how this can be achieved and the diversity of applications in which it can be employed, creating specific TC applications is often a difficult task.

This thesis concerns the design, implementation, and testing of an Object-Oriented Application Framework for Text Categorization. By encoding expertise in the architecture of the framework, many of the barriers to creating TC applications are eliminated. Developers can focus on the domain-specific aspects of their applications, leaving the generic aspects of categorization to the framework. This allows significant code and design reuse when building new applications.

Chapter 1 provides an introduction to automatic Text Categorization, Object-Oriented Application Frameworks, and Design Patterns. Some common application areas and benefits of using automatic TC are discussed. Frameworks are defined and their advantages compared to other software engineering strategies are presented. Design patterns are defined and placed in the context of framework development. An overview of three related products in the TC space, Weka, Autonomy, and Teragram, follows.

Chapter 2 contains a detailed presentation of Text Categorization. TC is formally defined, followed by a detailed account of the main functional areas

in Text Categorization that a modern TC framework must provide. These include document tokenizing, feature selection and reduction, Machine Learning techniques, and categorization runtime behavior. Four Machine Learning techniques (Naïve Bayes categorizers, k-Nearest-Neighbor categorizers, Support Vector Machines, and Decision Trees) are presented, with discussions of their core algorithms and the computational complexity involved. Several measures for evaluating the quality of a categorizer are then defined, including precision, recall, and the F_β measure.

The design of a framework that addresses the functional areas from Chapter 2 is presented in Chapter 3. This design is motivated by consideration of the framework's audience and some expected usage scenarios. The core architectural classes in the framework are then presented, and Design Patterns are employed in a detailed discussion of the cooperative relationships among framework classes. This is the first known use of Design Patterns in an academic work on Text Categorization software. Following the presentation of the framework design, some possible design limitations are discussed.

The design in Chapter 3 has been implemented as the `AI::Categorizer` Perl package. Chapter 4 is a short discussion of implementation issues, including considerations in choosing the programming language. Special consideration is given to the implementation of constructor methods in the framework, since they are responsible for enforcing the structural relationships among framework classes. Three data structure issues within the framework are then discussed: feature vectors, sets of document or category objects, and the serialized representation of a framework object.

Chapter 5 evaluates the framework from several different perspectives on two corpora. The first corpus is the standard Reuters-21578 benchmark corpus, and the second is assembled from messages sent to an educational ask-an-expert service. Using these corpora, the framework is evaluated on the measures introduced in Chapter 2. The performance on the first corpus is compared to the well-known results in [50]. The Naïve Bayes categorizer is found to be

competitive with standard implementations in the literature, and the Support Vector Machine and k-Nearest-Neighbor implementations are outperformed by comparable systems by other researchers. The framework is then evaluated in terms of its resource usage, and several applications using `AI::Categorizer` are presented in order to show the framework's ability to function in the usage scenarios discussed in Chapter 3.

Acknowledgments

I would like to thank Rafael Calvo for his expert supervision of my thesis, and for giving me the opportunity to pursue this project. The rest of the Web Engineering Group—Jae-Moon Lee, Xiaobo Li, Nick Carroll, and Gosia Mandrela—provided valuable testing and feedback on `AI::Categorizer` and created a quite pleasant research environment. The Language Technology group—particularly Casey Whitelaw, Elisabeth Crawford, and Jon Patrick—was also a good source of feedback and inspiration. The Open-Source community provided incentives to write clean, usable, documented software by their mere existence. Sheri Schechinger proofread this document and helped make countless readability improvements. Research Assistantship funding from the Capital Markets Co-operative Research Centre provided much-appreciated support for research on financial corpora.

On a more personal level, I would like to thank Sheri Schechinger for sticking with me across ten thousand miles of ocean, and the city of Sydney for being such a great place to spend time.

Preface

This thesis is the culmination of a Masters project in the Web Engineering Group at the University of Sydney School of Electrical and Information Engineering. The project has produced two large products—one is this thesis, and the other is the `AI::Categorizer` framework itself, which forms the subject matter of most of the thesis.

In order to produce such a framework, research into current Text Categorization algorithms has been necessary, as well as research into software engineering practices for building object-oriented frameworks. The discourse in this thesis does not assume any prior familiarity with Text Categorization, but it does assume that the reader is familiar with the basic concepts and terms of object-oriented programming, such as "class," "object," and "instance."

Availability

The latest released version of the `AI::Categorizer` framework (currently 0.04) is always available at `http://search.cpan.org/author/KWILLIAMS/`. Perl source code, documentation, and a simple example application are included in the distribution.

For developers who wish to stay more actively involved with tracking changes in the framework, the entire distribution is also available using the Concurrent Versions System (CVS). This allows developers to access the latest bug fixes, to create their own patches against the main framework code, and to track

changes between releases. Details of how to access the CVS version are at `http://sourceforge.net/cvs/?group_id=62831`, or via the project's development home page at `http://sourceforge.net/projects/ai-categorizer/`.

The ApteMod data set discussed in Chapter 5 is available for download from `http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.html`.

The Dr. Math data set is not available for direct download, but interested parties may contact Ken Williams at `ken@mathforum.org` for details.

After submission to the University of Sydney, this thesis document will be available in electronic format at `http://www.ee.usyd.edu.au/~kenw/Thesis.pdf`, and in hardcopy format from the University of Sydney Engineering Library.

Licensing

The `AI::Categorizer` framework is implemented as a set of Perl modules (see Section 4.1). As is customary with many Perl modules, the framework is distributed under the same licensing terms as the standard version of the Perl interpreter. This means that the user may choose either the GNU General Public License or the Artistic License as the terms of using the software, whichever fits better with their needs. In practical terms, this means that the code is encouraged to be used in research, commercial, educational, or other environments, without the need to pay royalties to the software's original author. It also means that the software's inner workings are available to be inspected or modified by other developers for their own projects.

Licenses of the above type are called "open source" licenses. Their goal is to foster the development and evolution of software by leveraging the user community and developer community as a resource that can feed back into the development cycle. According to `http://www.opensource.org/`, "open source promotes software reliability and quality by supporting independent peer review and rapid evolution of source code." This aligns very well with the traditional goals of academic research. By making the source code discussed in academic

publications available as open source resources, the results can far more easily be verified by other researchers.

For more information on open source concepts, please visit `http://www.opensource.org/`.

This thesis is copyright ©2003 by Ken Williams. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at `http://www.opencontent.org/openpub/`).

Motivations

My own personal motivations for embarking on this project were to further educate myself on Text Categorization research, to learn more about framework methodology, and to provide a software resource to others who wish to use TC methods in their software projects. I have long been interested in Machine Learning methods for various purposes, and I enjoy working with natural languages. Doing corpus-based work in Natural Language Processing is a fun combination of Machine Learning and Linguistics, and reading the literature on the topic always makes me excited to work on my next project.

Unfortunately, when I was just beginning to do work on my own Text Categorization projects, I found that there were very few TC tools freely available for my use, and those that were available were often difficult to customize. Very few tools were available in Perl, my usual language of choice, and this seemed like an odd situation given the well-known agility of Perl at handling text data. I cannot hope to solve everyone's software needs in TC, but the `AI::Categorizer` framework represents my best effort in providing the kind of thing I was looking for when I began working in this area.

My interest in framework development has recently increased by working on the `HTML::Mason` project [33]. For this, I and others helped shepherd the code from a fairly monolithic function-based tool to a customizable OO framework

suitable for many more purposes than it was originally developed to serve. I became convinced of the power of framework development with that project, and I sought to bring the same benefits to an open-source framework for TC.

Contributions

During the course of the candidature on which this thesis is based, the following contributions were accomplished:

- The `AI::Categorizer` framework was designed, implemented, and released under an open-source license [42]. The release includes documentation and a simple example application using the framework.
- Naïve Bayes and Decision Tree categorizers were implemented, as well as a mechanism which allows users to use categorizers implemented in the Weka Machine Learning system [47]. A simple probabilistic guessing categorizer has also been implemented to provide a baseline for experimentation.
- The framework currently has a Document Frequency feature selection module implemented.
- A paper on the design and applicability of the `AI::Categorizer` framework was published in the proceedings of the 7th Australasian Document Computing Symposium [44].
- A short paper on the use of the `AI::Categorizer` framework to categorize financial documents was published in the proceedings of the 7th Australasian Document Computing Symposium [5].
- A paper on the use of `AI::Categorizer` to automatically categorize mathematics questions will be published in the proceedings of the 11th International Conference on Artificial Intelligence in Education [45].
- New testing corpora have been assembled in the educational and financial domains, and the framework has been evaluated using them (see Chapter

5).

- Contributions from other developers have provided the framework with an SVM categorizer. Collaborative work with other developers have provided Rocchio and k-Nearest-Neighbor categorizers.

- An overview seminar on TC and the design of `AI::Categorizer` was given at the University of Sydney. An invited presentation of the same seminar was given to the Language Technology group at Macquarie University.

- Tutorials on Machine Learning were presented at the O'Reilly 2002 Open Source Conference and 2003 Bioinformatics Technology Conference (`http://conferences.oreilly.com/`).

Contents

Abstract **i**

Acknowledgments **iv**

Preface **v**

List of Figures **xiv**

List of Tables **xv**

1 Introduction **1**

1.1 Automatic Text Categorization 1

1.2 Object-Oriented Application Frameworks 3

1.2.1 Design patterns 5

1.3 Related products 6

1.3.1 Weka 7

1.3.2 Autonomy Corporation 8

1.3.3 Teragram Corporation 9

2 Background: Text Categorization **11**

2.1 Formal Definitions 11

2.2 The Text Categorization Process 13

2.2.1 Document storage 13

2.2.2 Document format 14

2.2.3 Document structure 14
2.2.4 Tokenizing of data 15
2.2.5 Dimensionality reduction 16
2.2.6 Vector space modeling 18
2.2.7 Machine Learning algorithm 19
2.2.8 Machine Learning configuration 19
2.2.9 Incremental learning 20
2.2.10 Hypothesis behavior 20
2.3 Machine Learning techniques 21
2.3.1 Naïve Bayes 21
2.3.2 k-Nearest-Neighbor 23
2.3.3 Support Vector Machines 24
2.3.4 Decision Trees 25
2.4 Performance Measures 25
2.4.1 Combining Measures Across Categories 28

3 `AI::Categorizer` Framework Design 30

3.1 Audience 31
3.1.1 Researcher 31
3.1.2 Application Developer 32
3.1.3 Domain Expert 32
3.2 Use Cases 33
3.2.1 Scientific investigations 33
3.2.2 Embedded applications 34
3.2.3 Client-server applications 34
3.2.4 Database cooperation 35
3.3 Overview of `AI::Categorizer` class hierarchy 35
3.4 Framework classes 38
3.4.1 KnowledgeSet 39
3.4.2 FeatureSelector 39

3.4.3 Collection . . . 40
3.4.4 Document . . . 41
3.4.5 Category . . . 42
3.4.6 Learner . . . 43
3.4.7 FeatureVector . . . 43
3.4.8 Hypothesis . . . 44
3.4.9 Experiment . . . 44
3.4.10 AI::Categorizer . . . 45
3.5 Design Patterns in `AI::Categorizer` . . . 46
3.5.1 Iterator . . . 47
3.5.2 Composite . . . 49
3.5.3 Adapter . . . 51
3.5.4 Strategy . . . 53
3.5.5 Factory Method . . . 56
3.6 Examples . . . 61
3.7 Limitations . . . 63
3.7.1 Structured Feature Vectors . . . 63
3.7.2 Hierarchical Categorization . . . 64

4 Implementation **66**
4.1 Implementation Language . . . 66
4.2 Framework constructor methods . . . 68
4.3 Data Structures . . . 69
4.3.1 Feature Vectors . . . 69
4.3.2 Sets of Documents or Categories . . . 71
4.3.3 Saving state . . . 72

5 Evaluation **73**
5.1 Corpora . . . 73
5.1.1 ApteMod . . . 74
5.1.2 Dr. Math . . . 75

5.2 Quality of Categorization 77

5.2.1 ApteMod 78

5.2.2 Dr. Math 80

5.3 Efficiency 81

5.4 Applications 83

5.4.1 Command-line categorizer 84

5.4.2 Database categorization 85

5.4.3 Client-server categorization 85

6 Conclusion **87**

6.1 Evaluation and Outcomes 87

6.2 Further Work 88

Bibliography **90**

Appendix A: A Framework for Text Categorization **95**

Appendix B: Automatic Categorization of Announcements on the Australian Stock Exchange **103**

Appendix C: Automatic Categorization of Questions for a Mathematics Education Service **107**

List of Figures

2.1 The action of a categorizer on a set of documents 12

3.1 Diagrammatic notation for object relationships 36

3.2 Inheritance diagram for `AI::Categorizer` 36

3.3 Class composition diagram for `AI::Categorizer` 38

3.4 The Iterator pattern in the `Collection` class 48

3.5 The Composite pattern in the `Learner::Ensemble` class 50

3.6 The Adapter pattern in the Learner class 52

3.7 The Strategy pattern in the Learner class 54

3.8 The Strategy pattern in the FeatureSelector class 55

3.9 A client-side approach to object construction 57

3.10 A framework-side approach to object construction 58

3.11 A centralized approach to object construction 59

3.12 Highest-level interface to `AI::Categorizer` 61

3.13 Separate invocations of experimental phases 62

3.14 Using `AI::Categorizer` for direct categorization of documents . 62

4.1 An example of `Class::Container` usage from the `Learner` class 68

5.1 Category distributions for the test corpora 75

5.2 Parameter specification file for testing the Naïve Bayes categorizer on the Dr. Math corpus. 85

List of Tables

2.1 Contingency table for category c_j and term f_k 17

2.2 Contingency table for category c_i 26

5.1 Results of `AI::Categorizer` on ApteMod corpus 78

5.2 Results from [50] on ApteMod corpus 78

5.3 Results of `AI::Categorizer` on Dr. Math corpus 80

5.4 Time required for the experiments in Section 5.2 82

5.5 Memory usage for the experiments in Section 5.2 82

Chapter 1

Introduction

1.1 Automatic Text Categorization

The field of automatic Text Categorization (TC) is an extremely active area of current research and application. It is multi-disciplinary, attracting attention from the Linguistics, Computer Science, Engineering, and Business communities. Its applicability is broad, with many potential uses for large businesses as well as individuals. A recent survey article from the Association of Computing Machinery provides a good introduction to the field [38].

The goal of automatic Text Categorization is to create systems that can automatically place text-based documents into predefined categories. For example, one system may assign themes such as "sports," "finance," or "politics" to general-interest news stories. Another system may automatically route a user's email messages by placing documents into folders based on the messages' content. In these scenarios, the news story or email message plays the role of a "document," and the news theme or email folder plays the role of a "category." TC systems' categorization decisions are usually based on some analysis of the words in each document, though they may be based on any arbitrary document properties.

The standard modern approach to TC involves using Machine Learning to

create categorizers automatically rather than manually specifying the membership criteria for each category [38, p. 2]. The Machine Learning process typically examines a set of documents which have been pre-assigned to categories, and makes inductive abstractions based on this data that will assist it in categorizing future documents [27, sec. 2.7].

Because the process of creating categorizers is automatic, and the categorization process itself is also automatic, efficient TC systems requiring no human intervention can be created that process large numbers of documents very quickly. In practice, human intervention may sometimes be applied in either phase, because manual tuning of the parameters that govern the creating of a categorizer may improve its performance, and because a human expert may assist the categorizer in making decisions, or vice versa.

The following quotation from [38] provides a sense of the broad range of applications currently using TC methods:

> TC is now being applied in many contexts, ranging from document indexing based on a controlled vocabulary, to document filtering, automated metadata generation, word sense disambiguation, population of hierarchical catalogs of Web resources, and in general any application requiring document organization or selective and adaptive document dispatching.

Because of the recent explosion in volume of electronic data due to the advent of the World Wide Web and the widespread use of email for business and personal communication, many new applications may benefit from using TC methods. Two application areas investigated during the course of this candidature include the use of TC methods to determine potential market impact of corporate financial announcements [5], and to assist educational mentors in managing a stream of messages sent to a mathematics question-and-answer service [44]. These tasks are currently performed by humans with special knowledge about the particular relationships between documents and categories, and

any gains in efficiency brought by automation may significantly aid the business processes of such organizations.

1.2 Object-Oriented Application Frameworks

An *Object-Oriented Application Framework* (hereafter referred to simply as a *framework*) is a large-scale unit of reusable code in object-oriented software development.[1] The relevant software engineering literature contains several different definitions of the term, with two definitions appearing most commonly:

- a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact [23]
- a reusable, "semi-complete" application that can be specialized to produce custom applications [14]

These definitions are not in conflict, but rather emphasize different aspects of framework development—the first definition emphasizes what a framework is made of, while the second emphasizes what a framework is used for. Notice that the first definition refers to the system's *design*, while the second refers to the system's actual *code*. This is because frameworks represent both code reuse and design reuse. The design of a system gets reused because any application built using the framework will embody the design decisions encoded in the framework structure, and the system code gets reused by employing the concrete classes provided with the framework [18, ch. 1].

In designing frameworks, developers strive to create a product that is useful in a maximum number of situations with a minimum of effort by the application developers. This is evidenced by the consistent appearance of the word "reusable" in the definitions in the previous section. In order to achieve effective reuse, the framework developer must identify those aspects of the target applications that vary from one application to the next, typically called *hot spots*,

[1] This thesis assumes that the reader is familiar with the basic terminology of object-oriented programming. For an introduction to the subject, please see [46] for an academic treatment, or [8] for an applied treatment.

and allow explicitly for their variations to be instantiated in applications [15, ch. 14] [10].

In some cases, the hot spot variations are known in advance to the framework developer, so concrete classes may be provided to the application developer to fulfill the variation requirements. Application development then becomes a simple matter of selecting the appropriate concrete classes for the application. This is known as *blackbox* framework usage.

In other cases, the application developer may have a particular need that the framework developer did not or could not anticipate. Application development then involves writing custom subclasses of the hot spot classes, a process known as *whitebox* framework usage [15, ch. 1].

Because blackbox framework usage involves much less effort than whitebox usage, most framework developers aim to provide blackbox functionality whenever possible. Since framework requirements may not be clear until the framework has been used in several different applications, however, many frameworks evolve from being primarily whitebox frameworks to primarily blackbox frameworks as they mature [18, ch. 6].

Frameworks are certainly not the only kind of software reuse technique in active use. Other reuse techniques include *components*, *libraries*, and *application generators*. A component is an element of a software system that can be replaced by other elements with similar purpose but different behavior [23]. A library is a set of routines or objects (possibly components) that provide functionality developers may use in application code [15, ch. 1]. An application generator is a system that creates varying applications based on high-level, domain-specific languages that specify the desired behavior of the application in its aspects that vary (i.e., in its hot spots) [15, ch. 1].

The biggest difference between frameworks and the above reuse techniques is that frameworks create an *inversion of control* between the framework code and the application developer code. Framework code assumes control of the main flow in an application, and any custom developer code (if the framework

is being used in a whitebox development style) is invoked by the framework. In the other reuse techniques mentioned above, the developer writes the high-level application code (whether in a programming language or in the application generator's mini-language) and invokes the reusable elements at a lower level. The inversion of control in frameworks lets the framework developer dictate the overall structure of the application, while allowing the application developer low-level control over application details [15, ch. 1].

According to [15, ch. 1], framework methodology offers the following benefits as a reuse technique:

Modularity The hot spots of a framework represent encapsulated solutions to the variations in the application domain. This helps minimize the impact of design and implementation changes in applications because they will usually be limited to these encapsulated areas.

Reusability Frameworks represent both design reuse and code reuse, leveraging both the expertise of the framework developer encoded in the framework architecture, and well-tested implementations encoded in the framework's concrete classes. In the case of blackbox usage, reuse may be achieved with no custom application code.

Extensibility Whitebox reuse allows frameworks to be used for purposes that the framework developer did not or could not foresee, and allows application developers to create interfaces to proprietary or non-generic entities while using the framework's general architecture.

Inversion of control Custom application code can play a subordinate role to generic framework code, so that different applications developed using the same framework will behave in similar ways at the highest levels.

1.2.1 Design patterns

In order to shed light on the design of complex object-oriented systems, many researchers and software developers have tried to standardize language, concepts,

and notation for class and object relationships. There is as yet no universally accepted terminology for describing these relationships, but one common practice is to use *design patterns* to provide a baseline grammar for discussing commonly seen patterns of cooperation in object-oriented design [18, p. 3]. The design patterns do not provide prescriptions for software design, but rather descriptions of common practices in common situations. Each design pattern in [18] includes discussions of variations that can be made in applying the pattern, indicating that a design pattern is actually a family of similar solutions to a problem, not one rigid solution.

Design patterns help to illustrate object-oriented software designs that use *composition* rather than just *inheritance* for embodying important relationships between objects. Composition refers to the practice of multiple independent objects cooperating to achieve a task, or assembling to form a larger functional unit, while inheritance refers to the practice of defining a single object's structure and behavior in terms of both general ("parent") and specific ("child") specifications. In the language of framework design and reuse, composition allows for blackbox reuse, while pure inheritance forces whitebox reuse [18, p. 19].

The relevance of several design patterns to `AI::Categorizer` will be discussed in detail in Chapter 3.

1.3 Related products

To discuss the relevance of `AI::Categorizer` in the marketplace of Text Categorization, three related products are examined here. These products are by no means the only available products similar to `AI::Categorizer`, but they provide a reasonable sample of well-known tools for comparison.

1.3.1 Weka

Weka is an open-source system for Machine Learning originally developed at the University of Waikato, New Zealand, by Ian H. Witten and Eibe Frank [47]. Its primary audience is the international community of academic Machine Learning researchers, most notably those working with categorization or clustering problems that arise from working with text. Weka has undergone at least one major code rewrite; at present it is implemented as a set of related Java classes with documented internal interfaces. Since these classes may be extended, Weka may itself be considered a framework.

Weka is used extensively throughout the academic Text Categorization community, and as such includes support for many cutting-edge categorization techniques. These include recent advances in Support Vector Machines, k-Nearest-Neighbor, Naïve Bayes, and other categorizers (see Section 2.3), as well as several variations of feature selection techniques (see Section 2.2.5). Weka therefore provides a standard against which the `AI::Categorizer` framework can be measured, as well as a resource which can be leveraged in its construction.

Despite some similar properties, Weka and `AI::Categorizer` differ in their goals and in many important implementation decisions. Whereas Weka specifically targets the academic research community, `AI::Categorizer` aims to support use cases under both application-building and research situations. Consequently, Weka will typically keep up with research trends more closely, but `AI::Categorizer` will usually be easier for application developers to integrate into a real-world situation.

In addition to these differences, another important difference arises from the different goals in the two projects. Much of the academic community is interested in evaluating the correctness and algorithmic complexity of categorization techniques, whereas most application developers must also consider resource usage in real-world terms like time and memory. In informal testing, `AI::Categorizer` has greatly outperformed Weka in terms of speed and memory when equivalent algorithms are compared on identical data sets. This doesn't reflect

an inherent design flaw in Weka, rather a difference in the kinds of things Weka developers are likely to spend their time working on.

In order to help facilitate cooperation between the Weka and `AI::Categorizer` communities, as well as leverage existing solutions inside `AI::Categorizer`, a machine learner class has been created within `AI::Categorizer` that simply passes data through to Weka's categorizers. In this way, application developers can easily experiment with Weka's cutting-edge categorization techniques while retaining `AI::Categorizer`'s application integration advantages. Any cross-pollination generated as a result will likely benefit both projects. See Section 3.6 for more information on the existing bridge to Weka.

Some other facilities included in Weka's distribution are not yet offered by `AI::Categorizer`. These include visualization tools and several sophisticated correctness evaluation tools. Most of these facilities would make useful additions to `AI::Categorizer` if implemented.

1.3.2 Autonomy Corporation

Autonomy Corporation (`http://www.autonomy.com/`) provides information services and product licensing to enterprise-level organizations. Some of its customers include General Motors, Ericsson, Sybase, Deutsche Bank, and the United States Department of Homeland Security. Its products range broadly over several areas of Text Processing and Information Retrieval, including categorization, summarization, and search systems. The company's web site claims that their products are "automatic, language independent, fast, scalable, and accurate." Since the products are proprietary, no independent verification of these claims has been done in this thesis, but the claims do provide a list of attributes this company feels are important in marketing its products.

The Autonomy web site indicates that its products utilize "Bayesian Inference and Claude Shannon's principles of information theory." While further details are not provided and the proprietary nature of Autonomy's products precludes much further analysis, this statement would be consistent with Naïve

Bayes categorization and an Information Gain feature selection criterion (see Sections 2.3 and 2.2.5). However, one must be cautious in making assessments like this, since there are other ways of employing Bayesian techniques for categorization, and Shannon's information theory pervades many areas of TC and Information Retrieval, including Decision Tree construction [31, 43] and search relevance ranking [20].

Autonomy suggests that their products can be useful in building customized portals, Customer Relationship Management (CRM) systems, enterprise-level search systems and document management tools, and Human Resources solutions. These are commonly encountered applications mentioned (but seldom illustrated) in the TC literature, and it seems to be generally felt that TC technologies apply broadly to these application areas.

1.3.3 Teragram Corporation

According to their web site (`http://www.teragram.com/`), Teragram Corporation is a provider of "fast and stable linguistic technologies, information search and extraction, knowledge management, and text processing technologies." One of their largest-scale products is the Teragram Categorizer, an automatic document categorizer that plays a similar technical role to `AI::Categorizer`. It cooperates with the Teragram Taxonomy Manager, which provides a user interface to categories and the documents within each category.

Like with Autonomy Corporation, Teragram's software products are proprietary, so little information on implementation is available. However, product capabilities and roles can be assessed from the marketing information given on the web site. The information presented here has all been gathered this way.

The Taxonomy Manager is a browser of hierarchical categories, similar to several on-line directory services like Yahoo (`http://www.yahoo.com/`) or the Open Directory Project (`http://www.dmoz.org/`). It might therefore be inferred that the Teragram Categorizer is a native hierarchical categorizer (see Section 3.7.2), or perhaps that it actually flattens the tree structure of the cat-

egory hierarchy into a flat list of its leaves, and imposes the tree structure only afterwards. Whichever case is true, it must be noted that the interfaces of the categorizer allow hierarchical categorization even if the internal workings are flat.

Another interesting aspect of Teragram's categorization technology is their Rule-Based Categorizer. Using this system, "each category within the directory is associated with a set of rules that describe documents within that category." This may be motivated by a need to integrate older hand-maintained lists of rules into newer applications, or it might be meant to address situations like email categorization in which most documents are indeed best categorized by simple rules (usually because the sender and receiver have agreed upon a tagging scheme to mark documents' important properties). It's not clear whether Teragram's Rule-Based Categorizer and Automatic Categorizer can cooperate on a single taxonomy, but they indicate that the two systems are complementary rather than exclusive.

Teragram also offers separate licensing for many of the tools that make up its products. In this sense, it has a strategy similar to one employed in `AI::Categorizer`'s design, in which useful pieces of functionality created for `AI::Categorizer` should be split off into their own products whenever possible.

Chapter 2

Background: Text Categorization

This chapter gives an overview of Text Categorization's terminology, methodology, and common contexts. Section 2.1 provides formal definitions of the foundations of TC methods, and the terms defined in this section will be used throughout the rest of this thesis. Section 2.2 introduces several aspects of TC that an application developer or researcher may need to control in a TC application or experiment. Section 2.3 discusses three machine Learning techniques common in TC, and Section 2.4 defines some typical ways of evaluating the performance of a TC system.

2.1 Formal Definitions

The goal of automatic Text Categorization is to automatically produce specialized functions that can process natural-language documents, assigning zero or more user-defined labels to them based on their content [38, p. 3] [26, ch. 16] [27, sec. 6.10]. More formally, given a set of labels (i.e., categories) $\mathcal{C} = \{c_1, \ldots, c_{|\mathcal{C}|}\}$ and a set of previously unseen documents $\mathcal{D} = \{d_1, d_2, \ldots\}$, a categorizer is a function $\mathcal{K}$ that maps from $\mathcal{D}$ to the set of all subsets of $\mathcal{C}$. Figure 2.1 shows a

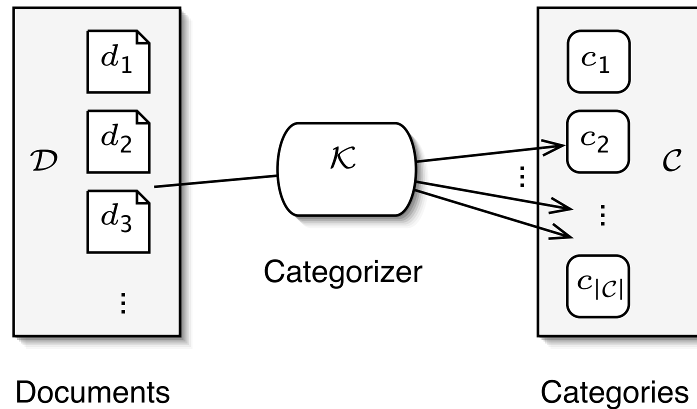

Figure 2.1: The action of a categorizer on a set of documents

simple diagram of this action.

In some applications, categorizers assign only a single label to each document, so a categorizer is often a function that maps directly from $\mathcal{D}$ to $\mathcal{C}$ [38, p. 3]. Often an intermediate function is useful for *soft* or *rank-based* categorization, mapping from $\mathcal{D} \times \mathcal{C}$ to the set of real numbers $\mathbb{R}$ in order to assign a score to each category c_j for each document d_i [38, p. 4]. The scored categories may then be presented to a human expert in decreasing order, and the human may then make the final decision on the document's category membership. Alternatively, the system may make a decision itself based on a threshold for category membership, transforming the problem back into the *hard* categorization shown in Figure 2.1.[1]

The standard modern approach to creating new categorizer functions is to build them using Machine Learning techniques from a set of training documents $\mathcal{T}^r$ [38, p. 2]. This is a set of user-provided, pre-labeled documents that follows a category distribution similar to the distribution of $\mathcal{D}$, and whose contents provide information about what sorts of documents should be mapped to what sorts of categories. Algorithms can then be developed that make generalizations about the relationship between document content and document category,

[1] This is the internal approach taken by the `AI::Categorizer` framework—see the description of the `Hypothesis` class in Section 3.4 for more details.

encoding these generalizations in the learned function $\mathcal{K}$.

2.2 The Text Categorization Process

In order to train a categorizer in the above manner, the user must begin with a *training corpus*, hereafter referred to as $\mathcal{T}^r$. This is a set of documents which are pre-labeled with categories that are considered to be fully correct—typically, they have been manually assigned by a *domain expert*, i.e. a person who is familiar with the type of material contained in the documents and who knows how to assign a set of categories to each document based on the documents' content.

The basic outline for creating Text Categorization applications is relatively simple: the documents in $\mathcal{T}^r$ are presented to the TC system, the system processes the documents' content, and a specific categorization function $\mathcal{K}$ is produced that may be used to categorize future documents from the set $\mathcal{D}$. In an application, however, many details of this process need to be managed in specific and often varying ways. Sections 2.2.1 through 2.2.10 describe the stages of this process.

2.2.1 Document storage

In an organization that needs a TC application, documents may have many different origins. They may originate from plain-text or formatted email messages, they may be raw or pre-processed web pages, they may be collections of data from a database (see Section 3.2.4), or they may not have a straightforward representation outside of the TC system at all. It is therefore important to recognize that the notion of varying document storage media, and the process of converting from those media to a medium accessible to the TC system, is an important part of the TC process.

In addition, many Text Categorization data sets are quite large. In their raw format, they may commonly be larger than the amount of available memory on

the machine processing them. This has two important implications. First, converting the documents to a special storage format (for instance, as a set of files on the local filesystem) so that the TC system can access them may be impossible or undesirable for reasons of time, space, and/or data redundancy. Second, a mechanism that can deal with iterating through the native storage medium of the documents without reading all document data into memory is probably necessary in a TC system.

2.2.2 Document format

Although most of the academic TC literature considers a document to be a simple plain text string of data, this may rarely be the case in an application environment. Documents may be stored in many different formats, including plain text, HyperText Markup Language (HTML), Adobe's Portable Document format, (PDF), Extensible Markup Language (XML), Microsoft Word .doc format, MIME-encoded email messages, and so on. The internal data in each document may also be considered part of its format when nontrivial amounts of information extraction or other transformations need to be applied to the document data in order to make it accessible to the TC system. For example, digit-strings in some document collections may be useful as terms to consider when categorizing, whereas in other collections they may only add noise to the data.

For reasons similar to those mentioned in the previous section, it may be desirable for a TC system to deal with these issues directly, or to provide a mechanism to extend the system to recognize new formats, rather than forcing the conversion of all data to a format recognized by the system.

2.2.3 Document structure

Separate from the issue of document *format* is that of document *structure*. In an age when XML data is increasingly more common as a data exchange and storage format, the structure of a document, i.e. the way the constituent

parts of a document interrelate and nest to create the entire document, may be important in understanding the document's meaning.

In the TC literature, little is currently made of document structure, except that a TC system may assign importance weights to the terms in a document according to pre-set importance weights of the sections in which those terms were found. For instance, a term found in the title of a document might be considered twice as important as a term found in the body. However, as research into categorization of structured documents progresses, this may be an fruitful area to consider in building TC systems.

2.2.4 Tokenizing of data

In order to convert the text of a document into data that may be analyzed by a Machine Learning algorithm, it is necessary to break the text into discrete units, each usually corresponding to a word in the text. This is called *tokenization*. In this discussion, the term *word* refers to a linguistic entity exactly as it appears in the original text, and *token* refers to a string extracted by the TC system.

The segmenting of text data into chunks representing individual words may seem like a straightforward task, but in fact there are many variations on how this process can be performed [26, sec. 4.2.2]. It is not sufficient to split the data into tokens by using whitespace (spaces, tabs returns, and the like) as delimiters, because this does not deal with punctuation characters. It is also not obvious whether words with punctuated suffixes like *boy's* or *doesn't*, or hyphenated words like *ready-made* ought to be treated as one token or multiple tokens—this decision will usually need to be made with some knowledge of the document set $\mathcal{D}$. In addition, many non-European written languages such as Japanese or Korean do not contain spaces indicating divisions between words, so a sophisticated tokenizer may be required when dealing with languages like these.

The tokenization process may remove from the data any token in a pre-defined "stop list," which contains tokens that occur very commonly in the

domain (such as "the" or "and" in most English texts) and are assumed to contain little or no relevance to the categorization problem at hand [38, p. 15] [28].

In order to address these issues, a TC system needs to allow variations in the tokenizing process. This may involve adjusting a set of control parameters, or in some cases the application developer may need to write custom code to handle domain-specific cases.

2.2.5 Dimensionality reduction

Like many Language Processing research fields, much of Text Categorization has to do with the problem of *high dimensionality* [38, p. 13] [21]. The dimensionality of the space in which the Machine Learning algorithm operates can be as large as the total number of distinct terms in $\mathcal{T}^r$.

High dimensionality may present two problems. First, some Machine Learning algorithms may be efficient on low-dimensional data, but they may require more time or memory than is practical when the dimensionality of the data set is too high [6]. Second, the $\mathcal{T}^r$ data in a high-dimensional space may be too sparse, with not enough nonzero data points to make any useful inductive leap during training. This is particularly true in some highly morphological languages like Finnish, in which a single word stem may have thousands or millions of inflected forms, and most forms may only be seen one time in all of $\mathcal{T}^r$, making them almost useless for inductive learning.

One way to address the problem of high dimensionality is by applying a linguistic *stemming* algorithm to the terms found in $\mathcal{T}^r$ and $\mathcal{D}$. These algorithms transform words by removing prefix and suffix morphemes, so that words like *running* and *runner* collapse to their linguistic stem *run*. Although the use of such processing has occasionally been reported to harm overall system performance [1], availability of such an algorithm is usually seen as a necessary component of a TC system [38, p. 12].

Another way to reduce dimensionality in a TC system is to apply *feature*

		c_j **occurs**	
		Yes	No
f_k **occurs**	Yes	A	B
	No	C	D

Table 2.1: Contingency table for category c_j and term f_k. The quantities A-D represent the number of documents with the given properties.

selection and/or *feature extraction*. Both are statistical techniques to transform the entire set of document terms into a smaller feature set with less sparsity. The former does this by choosing the “most relevant” terms using some statistical criterion. The latter applies some transformation such as singular value decomposition (used in the “Latent Semantic Indexing” technique [9]) or grouping terms into clusters in order to create a new, lower-dimensional space of features.

Three of the most commonly used feature selection criteria are the *document frequency* (DF), χ^2, and *information gain* (IG) metrics [51].

DF(f_k) is simply the number of documents of $\mathcal{T}^r$ in which the feature f_k occurs. It is fairly effective as a feature selection criterion, and it has been found by [51] that other well-performing criteria have a bias favoring terms with high DF.

χ^2 is defined in Equation 2.1. A, B, C, and D are defined as the terms in the contingency table shown in Table 2.1. $\chi^2(f_k, c_j)$ has a value of zero when f_k and c_j are independent, and of 1 when they are perfectly correlated.

$$\chi^2(f_k, c_j) = \frac{|\mathcal{T}^r|(AD - CB)^2}{(A+C)(B+D)(A+B)(C+D)} \tag{2.1}$$

In order to find the overall $\chi^2(f_k)$ metric, the terms $\chi^2(f_k, c_j)$ may either be averaged (typically weighted by the frequency of each category), or the maximum value for any category may be adopted [51].

IG is defined in Equation 2.2. $P(f_k)$ and $P(\overline{f_k})$ are the probabilities that a document does or does not contain f_k, respectively. $\mathcal{C}_{f_k}$ and $\mathcal{C}_{\overline{f_k}}$ are the category sets in the subsets of $\mathcal{D}$ containing f_k or not, respectively. H(x) is the standard

entropy function from Information Theory [26, ch. 2].

$$IG(f_k) = \mathrm{H}(\mathcal{C}) - P(f_k)\,\mathrm{H}(\mathcal{C}_{f_k}) - P(\overline{f_k})\,\mathrm{H}(\mathcal{C}_{\overline{f_k}}) \tag{2.2}$$

Information Gain has a natural value of zero for non-informative features, and values increase for features that correlate more strongly with certain categories. It has been found in [51] to be a very effective feature selection criterion.

2.2.6 Vector space modeling

The discussion in the previous section suggests that each document may be viewed as a vector in a global vector space whose dimensions represent the set of all unique features from $\mathcal{T}^r$. This idea forms the basis for several Machine Learning techniques, including Support Vector Machines and k-Nearest-Neighbor categorizers. It also allows for arbitrary vector processing algorithms on the document data to improve categorization results.

A common set of algorithms used for this purpose in Information Retrieval is the *TF/IDF* term-weighting scheme of Salton and Buckley [36], which allows for several different ways to represent a document as a vector in the global vector space. Terms may be weighted by their frequency in the document, by the logarithm of that frequency, or by a boolean figure representing only the presence or absence of the term. Term weight may also be reduced by a factor representing the term's prevalence in other documents, on the theory that any term present in most corpus documents possesses little discriminatory power between categories. Finally, the overall length of the document vector may be scaled in several different ways. The TF/IDF vector weighting techniques are used commonly in TC systems, and their availability is desirable for the `AI::Categorizer` framework.

2.2.7 Machine Learning algorithm

Many different Machine Learning algorithms are actively studied in the TC research literature, and new algorithms or variations on existing algorithms are continually being developed. In addition, the choice of algorithm may depend on the specific application—algorithms differ not only in their ability to perform accurately on differing data sets, but also in the resources they may require during training and when categorizing documents. Therefore, it is not possible to choose a single Machine Learning algorithm for incorporation into the `AI::Categorizer` framework. As a TC system, it needs to allow for selection among several standard algorithms and for incorporation of novel algorithms developed by researchers.

Section 2.3 gives an overview of three well-studied Machine Learning algorithms and compares some of their relevant characteristics. Section 3.5.4 in Chapter 3 shows how the architecture of `AI::Categorizer` allows for flexibility in this aspect of categorization.

2.2.8 Machine Learning configuration

Even within a single Machine Learning algorithm there may be several parameters that a supervisor may vary to influence the training and categorization processes. For instance, the k-Nearest-Neighbor algorithm has an adjustable parameter k, the SVM algorithm allows for several variations in the type of kernel used, and most categorization algorithms admit some type of control to trade off precision and recall against each other (see Section 2.4 for an explanation of these terms). In order to achieve the appropriate performance for a given task, application developers need simple ways to vary these parameters.

In fact, this issue is not unique to the Machine Learning component of the TC process. Several of the previously discussed aspects of the TC task, including feature selection, dimensionality reduction, and vector space transformation, may be controlled by parameters that the supervisor may wish to vary. Con-

sistency in the system's handling of parameters may therefore be an important part of its design. This issue will be discussed again in Section 3.5.5.

2.2.9 Incremental learning

In some TC applications, it may be desirable to incorporate feedback from the user about whether the system's categorization decisions have been correct or incorrect [38, p. 28]. This may allow for a relatively small initial training set $\mathcal{T}^r$, or for categorization on concepts which may change over time. This process is called *incremental* or *on-line* learning.

Unfortunately, incremental learning is not possible with all Machine Learning methods, since some algorithms (e.g. Neural Network categorizers) may not be able to incorporate new evidence into their model without going through the entire training process again. For those algorithms which can support it, however, the use of incremental learning may be considered important in building a TC application, and is therefore considered a goal of the `AI::Categorizer` project.

2.2.10 Hypothesis behavior

Most of the standard Text Categorization literature assumes that the goal of TC is to assign each document to one of two mutually exclusive categories, otherwise known as *binary categorization* [38, p. 3]. Of course, real-world problems often involve ontologies (category hierarchies) that consist of more than two categories, and membership may not be mutually exclusive. For instance, some applications may require assigning only the most appropriate category from $\mathcal{C}$, some may require assigning any appropriate category, and some, such as rank-based tasks, may require an appropriateness score linking each category-document pair.

This situation does not represent a theoretical disconnect between research and practice, because each multi-category TC problem can be re-posed as a series of binary problems. Most application builders, however, will not want to

actually re-pose their problems in this manner because it requires extra work and it may obscure the true nature of the application under development. Therefore, it is desirable for a TC system to offer support for all the scenarios described in the previous paragraph without requiring the implementers of the Machine Learning algorithms to explicitly code for them. The way in which this is achieved is discussed in Section 3.3.

2.3 Machine Learning techniques

This section describes four Machine Learning techniques that are common for Text Categorization: Naïve Bayes categorizers, Support Vector Machines, k-Nearest-Neighbor categorizers, and Decision Trees. These techniques were chosen for inclusion here because they currently have implementations in the `AI::Categorizer` framework and because the first three are included in the well-known study [50]. Their performance is revisited in Chapter 5.

2.3.1 Naïve Bayes

NaïveBayes categorizers are extremely well-represented in the TC literature, with many papers published examining their theory and performance [25, 50, 38]. Their theory rests on Bayes' Theorem of conditional probability, shown in Equation 2.3. For those unfamiliar with conditional probabilities, the notation $P(a|b)$ means "the probability of a given b."

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} \tag{2.3}$$

The quantity of interest when determining the relevance of a particular document d_i to a category c_j is $P(c_j|d_i)$. Any category with a high enough conditional probability will be considered assigned to d_i. In particular, the "best" category will be $\mathrm{ArgMax}_{c_j \in \mathcal{C}}\, P(c_j|d_i)$. This probability is usually impossible to compute directly, however, because d_i has likely never been encountered before.

Therefore, Bayes' Theorem can be applied to change the probabilistic expression to one whose terms may be estimated from the training data $\mathcal{T}^r$ as follows.

$$\begin{aligned}\operatorname*{ArgMax}_{c_j \in \mathcal{C}} P(c_j|d_i) &= \operatorname*{ArgMax}_{c_j \in \mathcal{C}} \frac{P(d_i|c_j)P(c_j)}{P(d_i)} && \text{(by (2.3))} \\ &= \operatorname*{ArgMax}_{c_j \in \mathcal{C}} P(d_i|c_j)P(c_j) && (P(d_i) \text{ is constant})\end{aligned}$$

$P(c_j)$ may be easily estimated from the frequency with which documents appear in c_j in $\mathcal{T}^r$. To estimate $P(d_i|c_j)$, d_i may be considered equivalent to the string of its features $f_{i1}f_{i2}\ldots f_{ik}$. $\operatorname{ArgMax}_{c_j \in \mathcal{C}} P(c_j|d_i)$ may then be estimated as follows.

$$\begin{aligned}\operatorname*{ArgMax}_{c_j \in \mathcal{C}} P(c_j|d_i) &= \operatorname*{ArgMax}_{c_j \in \mathcal{C}} P(f_{i1}\ldots f_{ik}|c_j)P(c_j) \\ &\approx \operatorname*{ArgMax}_{c_j \in \mathcal{C}} P(f_{i1}|c_j) \cdot \ldots \cdot P(f_{ik}|c_j)P(c_j)\end{aligned}$$

This final step, which gives this algorithm its "naïve" moniker, involves two conditional independence assumptions: first, that the features $f_{i1}, \ldots, f_{ik}$ are conditionally independent given the category c_j, and second, that the position of features within document d_i has no effect on the probability. These features may not be true in general—features may in fact correlate in complex ways in real-world documents. Nevertheless, the Naïve Bayes categorizer tends to produce fairly good results, and an analysis of this phenomenon can be found in [12].

The conditional probabilities $P(f_{i1}|c_j), \ldots, P(f_{ik}|c_j)$ are typically estimated by measuring the relative frequencies of the features $f_{i1}, \ldots, f_{ik}$ in the documents belonging to c_j. For features found in the documents of $\mathcal{D}$ which were not encountered in $\mathcal{T}^r$, it would be inappropriate to use this estimate, however, because it would yield a probability of zero and render the rest of the terms useless. For this reason, unknown terms are typically assigned some small nonzero

probability in a process known as *smoothing*.

The Naïve Bayes algorithm is fairly fast and non-memory-intensive. Because its training process consists merely of counting the features of the training documents, its training time scales linearly with $|\mathcal{T}^r|$. Categorization is also fairly fast because all the pre-computed probabilities $P(f_{il}|c_j)$ may simply be looked up in an array. Categorization of a single document therefore scales linearly with $|\mathcal{C}|$. Because the system need only store feature information on a per-category basis instead of a per-document basis, the size of the trained categorizer will stay fairly small compared to more resource-intensive categorizers like k-Nearest-Neighbor.

2.3.2 k-Nearest-Neighbor

The k-Nearest-Neighbor algorithm (kNN) is one of the most conceptually simple TC algorithms in the literature. All documents in $\mathcal{T}^r$ are considered as vectors in a space with a similarity measure $m : \mathcal{D} \times \mathcal{D} \to \mathbb{R}$. To determine whether an unseen document d_i is assigned to a category c_j, the k most similar documents to d_i using the measure m are determined, where k is a user-adjustable parameter. If the number of these k documents that belong to c_j (possibly weighted by the similarity measure m for each similar document) is greater than some predefined threshold, then d_i is assigned to c_j, and otherwise not. This technique has been described in [38, p. 28], [50], and [51], among others.

The choices for the k parameter, the category-membership threshold, the similarity function m, and how to map from the similarity scores m to the overall score for c_j provide for many variations on the standard algorithm. For instance, a single membership threshold may be used for all categories, or a different threshold may be used for each category, possibly learned by optimizing performance on a validation set. In addition, if more than one document is being categorized in a batch operation, several differing strategies for thresholding may be employed that take advantage of the overall proportions of documents that belong to each category [49].

Although the k-Nearest-Neighbor algorithm is conceptually simple, it is computationally intensive. Unless thresholds are learned from a validation set, there is no actual training stage when building a categorizer—all decisions are made by computations performed during categorization. The time to train a kNN categorizer is therefore minimal or null, but the time to categorize a single unseen document scales linearly with $|\mathcal{T}^r|$ and must be performed in full for each categorization. In addition, the entire training corpus $\mathcal{T}^r$ must be preserved in the categorizer's model, so memory or storage requirements may be prohibitively high in some environments.

2.3.3 Support Vector Machines

Support Vector Machines (SVM) are another extremely active area of research in the Text Categorization literature. Their use in TC was introduced by [21], and several studies, including [21] and [50], have found their results to be highly competitive with other Machine Learning methods on the standard benchmark corpora.

SVM techniques are similar to kNN in that they view the training documents as vectors in a vector space, and that they require a similarity function (called the "kernel" function) that plays a role similar to the function m in Section 2.3.2 [37, ch. 1]. However, instead of considering the documents most similar to the document to be categorized, SVM algorithms learn a *decision surface* during training which divides the vector space into regions that indicate category membership. Categorization then simply consists of determining which side of the decision surface each document lies on.

One key advantage of SVMs is that they can deal well with very large feature spaces, both in terms of the correctness of the categorization results and the efficiency of the training and categorization algorithms. This implies that little or no feature selection may need to be performed on the training data, removing a possibly time-consuming aspect of the TC process. Unfortunately, a disadvantage of many SVM training algorithms is that they scale poorly with $|\mathcal{T}^r|$, in

some cases requiring as much as $O(|\mathcal{T}^r|^3)$ or $O(|\mathcal{T}^r|^4)$. This may make their use with large numbers of documents prohibitive unless the standard algorithms are modified.

2.3.4 Decision Trees

Decision Trees (DT), developed during the 1960s and applied to Text Categorization by [31], are a popular Machine Learning technique in the TC literature [27, ch. 3] [38, p. 22] [26, sec. 16.1]. DT algorithms involve the construction of a tree structure to be used in categorizing documents. Each node in the tree refers to a feature from the training corpus, each branch entails a test on the feature's weight in the given document, and each leaf indicates a category to assign to the document [38, p. 22].

The automatic construction of Decision Trees can be a difficult and time-consuming task. The main benefit provided by DTs is that the tree structure is easily interpretable by humans, making the categorizer's decision-making process transparent to the user. The tree may also be converted into a set of boolean rules [27, sec. 3.7.1.2], which may help a user further understand the process.

Because of a need to limit the scope in which testing was performed, and because [50] does not evaluate Decision Tree categorizers, the discussion in Chapter 5 does not include the Decision Tree categorizer in `AI::Categorizer`.

2.4 Performance Measures

Several statistical measures have become standard in the area of evaluating Text Categorization systems [38, p. 33]. Some of the most prevalent are based on the notions of *precision* and *recall* from the field of Information Retrieval [41]. Precision, often denoted by the symbol π, measures the probability that a document assigned by the TC system to a given category actually belongs to that category. Conversely, recall, denoted by ρ, measures the probability that

		Expert choice	
		Yes	No
System choice	Yes	A_i	B_i
	No	C_i	D_i

Table 2.2: Contingency table for category c_i

a document actually belonging to a certain category will be assigned during testing to that category [38, p. 33].

The probabilities mentioned above can be estimated during testing by comparing how often the TC system's category choices match the correct categories. A valuable tool for this analysis is the "contingency table," which summarizes the results of the experiment for a given category. Table 2.2 shows a contingency table for the category c_i, i.e. any arbitrary category in the categorization scheme of the corpus. Here, A_i, etc. represent the number of documents that fall into the given situation, i.e. A_i is the number of test documents assigned to category c_i by both the expert and the TC system.

This allows us to estimate π and ρ, whose true values are $P(Expert = Y|System = Y)$ and $P(System = Y|Expert = Y)$, respectively, in terms of the entries in the contingency table. Since the number of documents assigned to category c_i by the TC system is $A_i + B_i$, and the number assigned by the expert is $A_i + C_i$, our estimates for π and ρ are $\frac{A_i}{A_i+B_i}$ and $\frac{A_i}{A_i+C_i}$, respectively.

π and ρ give valuable information about the performance of a TC system, but neither provides an isolated rating of the system's quality. The reason is that either measure can usually be improved in a system to the detriment of the other [38, p. 35]. For instance, the *trivial acceptor* categorizer, which assigns every document to every category, will have a perfect ρ score of 1, but its precision will be unacceptably low on any nontrivial task.

Therefore, a measure that combines π and ρ is desirable as an overall measure of the quality of the TC system. One such measure is the F_β measure, first introduced to the Information Retrieval literature by van Rijsbergen [41, ch. 7]. It is defined by the equation $F_\beta = \frac{(\beta^2+1)\pi\rho}{\beta^2\pi+\rho}$, where $0 \leq \beta \leq \infty$. The β parameter

provides a continuous way to balance between the importance of π and ρ, with values closer to 0 emphasizing π, values closer to ∞ emphasizing ρ, and a value of 1 balancing the two measures equally. Without specific knowledge of an application's requirements (for instance, whether false positives for a certain category are more problematic than false negatives), one may presume that π and ρ are equally important, and therefore the literature often uses F_1 as a measure of the quality of a TC system on a particular category.

${F_1}_i$ may be derived in terms of the entries of the per-category contingency table as in Equation (2.4).

$$\begin{aligned} {F_1}_i &= \frac{2\pi_i\rho_i}{\pi_i + \rho_i} \\ &= \frac{\frac{2{A_i}^2}{(A_i+B_i)(A_i+C_i)}}{\frac{A_i}{A_i+B_i} + \frac{A_i}{A_i+C_i}} \\ &= \frac{2{A_i}^2}{A_i(A_i+C_i) + A_i(A_i+B_i)} \\ &= \frac{2A_i}{2A_i + B_i + C_i} \end{aligned} \qquad (2.4)$$

Two other measures of categorization quality, *error* and *accuracy*, are also sometimes encountered in the TC literature. These are simple measures which can also be defined in terms of the contingency table in Table 2.2: $error = \frac{B_i+C_i}{A_i+B_i+C_i+D_i)}$, and $accuracy = \frac{A_i+D_i}{A_i+B_i+C_i+D_i)}$. In other words, error is the proportion of the system's decisions that matched the expert's choices, and accuracy is the proportion that did not. As summarized in [38, p. 34], these are not always useful measures of categorization quality, because the *trivial rejector* (a system that never assigns any documents to any category) will often have a lower error and higher accuracy than most nontrivial categorizers. Nonetheless, error will be measured for the evaluation tasks here, because it may give insight into the character of the system's performance.

2.4.1 Combining Measures Across Categories

Section 2.4 introduced several performance measures that may be defined to evaluate a categorizer on a single category. In order to evaluate the categorizer's overall performance on the entire set of test documents it is desirable to combine the per-category scores π_i, ρ_i, and F_{1i} into overall scores for the entire category set.

Two methods for doing this are standard in the literature. The first is called *micro-averaging*, which sums the terms in the contingency table for all categories simultaneously rather than in per-category tables. In other words, the micro-averaged π, ρ, and F_1, notated π^μ, ρ^μ, and F_1^μ, are defined in terms of the per-category contingency tables by the equations in (2.5).

$$\pi^\mu = \frac{\sum_{i=1}^{|C|} A_i}{\sum_{i=1}^{|C|} A_i + B_i} \qquad \rho^\mu = \frac{\sum_{i=1}^{|C|} A_i}{\sum_{i=1}^{|C|} A_i + C_i} \tag{2.5}$$

$$F_1^\mu = \frac{\sum_{i=1}^{|C|} 2A_i}{\sum_{i=1}^{|C|} 2A_i + B_i + C_i}$$

Micro-averaging gives equal weight to each categorization decision made by the system, or equivalently, to each document in the corpus, regardless of how many categories it belongs to.

An alternative to micro-averaging is *macro-averaging*, in which the per-category scores π_i, ρ_i, and F_{1i} are simply averaged to find the macro-averaged π, ρ, and F_1, notated π^M, ρ^M, and F_1^M. The equations in 2.6 describe this procedure.

$$\pi^M = \frac{\sum_{i=1}^{|C|} \pi_i}{|C|} \qquad \rho^M = \frac{\sum_{i=1}^{|C|} \rho_i}{|C|} \qquad F_1^M = \frac{\sum_{i=1}^{|C|} F_{1i}}{|C|} \tag{2.6}$$

Macro-averaging gives equal weight to each category in the corpus, regardless of how many documents it contains. Thus it provides a good counterpart to

micro-averaging; macro-averaging will place more emphasis on rare categories than micro-averaging, so reporting both scores is typically useful to evaluate the system as a whole.

Because micro-averaging emphasizes performance on common categories, and categorizers will typically perform better on categories with more training examples, micro-averaged performance scores are usually higher than macro-averaged scores. The size of the gap between the micro- and macro-averaged scores can be a good indicator of the difference in performance of the system on common and rare categories.

Note that the error and accuracy measures are unaffected by micro- vs. macro-averaging, as shown in Equation 2.7. This uses the observation that $A_i + B_i + C_i + D_i = |\mathcal{T}^e|$, a consequence of the fact that exactly one of the terms on the left side will be incremented with each decision about whether or not a document from the test set belongs to c_i.

$$
\begin{aligned}
error^M &= \frac{\sum_{i=1}^{|\mathcal{C}|} error_i}{|\mathcal{C}|} \\
&= \frac{\sum_{i=1}^{|\mathcal{C}|} \frac{B_i+C_i}{A_i+B_i+C_i+D_i}}{\sum_{i=1}^{|\mathcal{C}|} 1} \\
&= \frac{\sum_{i=1}^{|\mathcal{C}|} \frac{B_i+C_i}{|\mathcal{T}^e|}}{\sum_{i=1}^{|\mathcal{C}|} 1} \\
&= \frac{\sum_{i=1}^{|\mathcal{C}|} B_i + C_i}{\sum_{i=1}^{|\mathcal{C}|} |\mathcal{T}^e|} \\
&= \frac{\sum_{i=1}^{|\mathcal{C}|} B_i + C_i}{\sum_{i=1}^{|\mathcal{C}|} A_i + B_i + C_i + D_i} \\
&= error^\mu
\end{aligned}
\tag{2.7}
$$

Chapter 3

`AI::Categorizer` Framework Design

Framework design is a difficult task in general, because a well-designed framework must allow for several kinds of growth [15, p. 11]. The framework interface must be usefully applied to several different use cases, including ones that the framework designer may not be able to foresee. The framework must also be extensible by subclassing, and must therefore have enough structure that the relationships among classes are well-defined, yet flexible enough that the application developer can make appropriate modifications.

In designing the `AI::Categorizer` framework, attention has been paid to three primary areas: the framework's audience motivates the interface, use cases motivate the functionality, and algorithms and data structures motivate the implementation. In this chapter the functionality and interface decisions will be discussed in detail. Implementation will be discussed primarily in Chapter 4. However, some implementation issues inevitably motivate design, so they will be mentioned in this chapter as appropriate.

For brevity, the `AI::Categorizer::` prefix will be omitted from class names in this discussion. It is to be understood that any class within the `AI::Categor-`

izer framework (except the top-level class AI::Categorizer itself) is prefixed by AI::Categorizer::.

3.1 Audience

The design process for any sufficiently complicated software system benefits from consideration of its users and the specific ways they will want to interact with the system [2, ch. 16]. This provides both a motivation for the design of the system's interfaces, and a way to evaluate the system during and after its development, by ensuring that the users can use the system to perform the required functionality.

The main users of Text Categorization software may be generally divided into three categories: TC researchers, application developers, and domain experts. Of course, one person may play several of these roles simultaneously, but it is helpful during the design process to separate these roles for analysis.

3.1.1 Researcher

A researcher is interested in exploring novel approaches to machine learning or document processing. This professional is often not interested in implementing a real-world application, but wishes to improve existing Text Categorization algorithms and methodologies.

A researcher will often extend the framework with custom code that implements new functionality. For instance, the researcher may implement new machine learning algorithms or variations on existing algorithms. Researchers will also need tools for comparing the results of categorization experiments and may find it convenient to have a graphical interface for running common kinds of experiments.

Although a researcher will often need to write low-level framework extension code, that code will often be called from a high level. A researcher's application programs may be extremely simple, in effect training a categorizer and testing

it on a set of test documents.

3.1.2 Application Developer

An application developer is a professional such as a web developer or engineer that needs to add automatic categorization features to a software application. The application developer may have no prior experience with Text Categorization, but may still need to control the TC process closely because of specific application needs. An application developer may want to treat a TC system as a library or set of libraries, providing no custom code of his or her own. Alternatively, the developer may add custom code for accessing data in the application's native formats or integrating with the application's environment.

While the application developer may write less custom framework code than the researcher, framework usage may be more complicated. The application developer is often interested in very specific aspects of the categorization process, such as which/how many categories are assigned to any given document. Thus the application developer will typically create more complex applications using the framework than the researcher, exercising the framework API to a greater extent.

3.1.3 Domain Expert

Complex applications often require a domain expert who dictates project requirements and has expertise in the application domain (for example, financial documents or knowledge management). The domain expert often makes high-level decisions about when Text Categorization could be effective in the given domain, and may need to exert fine control over the Text Categorization process. The domain expert may delegate actual software development to the other members of a business team. The domain expert may also be responsible for building and maintaining the training set $\mathcal{T}^r$ on which the performance of the TC system depends.

3.2 Use Cases

In order to better understand and document how the framework will be used, an analysis of use cases is often helpful [2, ch. 2]. Use cases provide details of the required functionality of a project. They can also provide a starting point for design of the project's architecture. In this section, several common use cases for a Text Categorization application are discussed. The design of the framework should be directed toward satisfying these use cases.

3.2.1 Scientific investigations

Much of the academic work on Text Categorization is scientific investigation into various techniques for document processing [38]. This work may include investigations into methods for preprocessing document content, feature selection and extraction, or machine learning methods. Most often, researchers will develop or adopt a measurement for the quality of results, then compare two or more document processing methods and present the measurements for each method.

A typical use case for this type of investigation is as follows. The researcher obtains one or more corpora of documents on which to perform his or her experiments. If the corpus data is not in a format compatible with the tools being used, the data must be transformed into a different format. The data is then loaded into one or more systems that process the data. In a research setting, at least one of these systems will likely have components developed by the researcher, as novel work is usually under investigation. The outcome of the systems' processing is then collected and analyzed using the measurement for quality of results, and the work is presented to others for review.

Variations on this use case may arise from the specific area under investigation. For instance, if the researcher is investigating feature selection, different elements of the TC software will be used or customized than if the researcher is investigating machine learning techniques. The process flow may also vary

depending on whether the researcher is repeating the same process many times on different data sets, different processes many times on the same data set, or using a different methodology.

In most cases, the researcher will also need a way to keep track of experimental procedures and settings so that results under different conditions can be compared. This functionality may be directly provided by a categorization framework, or it may be provided by application layers written on top of the framework.

3.2.2 Embedded applications

In order to be useful in real-world applications, a categorization framework may need to function in multiple kinds of embedded environments. For example, a web-based application might embed categorization functionalities directly in the web server, or a categorization-enabled database might embed a categorizer directly in the database engine. A TC framework that can exist in these environments will increase its usefulness.

3.2.3 Client-server applications

An alternative to the embedded applications described in Section 3.2.2 is to use a client-server model. In this model, the application developer creates a dedicated categorization server which communicates over a data socket with clients. The main application (such as the web server or database described above) communicates over a data socket with the categorization server. Recent standardizations in protocols such as SOAP [3] and XML/RPC [40] have provided commonly-available, easy-to-use tools for creating these kinds of applications. Since a single categorization server can provide services to multiple application clients, developers may reduce development time when building TC applications in this manner. In addition, using the client-server model allows organizations to separate the categorization system from the front-end application, which may be necessary when the document data is sensitive or proprietary.

3.2.4 Database cooperation

Since organizations often store important data in a relational database, a TC framework can provide important services by cooperating directly with the database. This cooperation may involve retrieving documents from the database, retrieving document-category membership information from the database, using the database as a storage medium for the learned categorization model, or providing categorization services to database queries in the form of SQL functions.

A fellow student at the University of Sydney, David Bell, has created an interface between the PostgreSQL relational database and the `AI::Categorizer` framework, allowing the retrieval of documents from the database and access to categorization functionality via SQL functions [45]. Although the SQL interface functions will need to be rewritten for each database system (e.g. Oracle, Sybase, MySQL), the core framework functionality can be reused.

3.3 Overview of `AI::Categorizer` class hierarchy

In order to understand the structure of the `AI::Categorizer` framework, multiple kinds of analysis are helpful. We can examine the inheritance relationships of the classes that participate in `AI::Categorizer` and indicate which classes inherit from each other. Since a class generally is a representation of certain responsibilities and capabilities, this lets us see how the set of responsibilities for one class may be implemented in different ways or extended by its subclasses.

Figure 3.1 explains the notational elements used in the diagrams in this section. Because [18] is heavily drawn upon throughout this chapter, a notation closely following its notation is used here, with some elements borrowed from common UML [2, ch. 4-5]. Abstract classes (classes in which key functionality is left undefined, and which must be subclassed before being used in an application) are represented using italic font faces, and concrete (non-abstract) classes are represented using bold font face.

Figure 3.2 shows the inheritance relationships among classes in the `AI::Cat-`

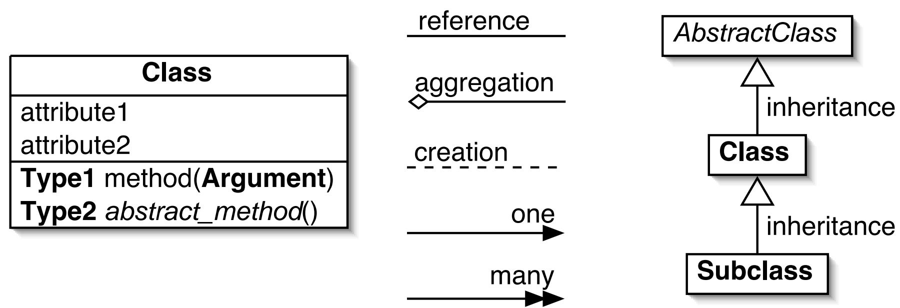

Figure 3.1: Diagrammatic notation for object relationships

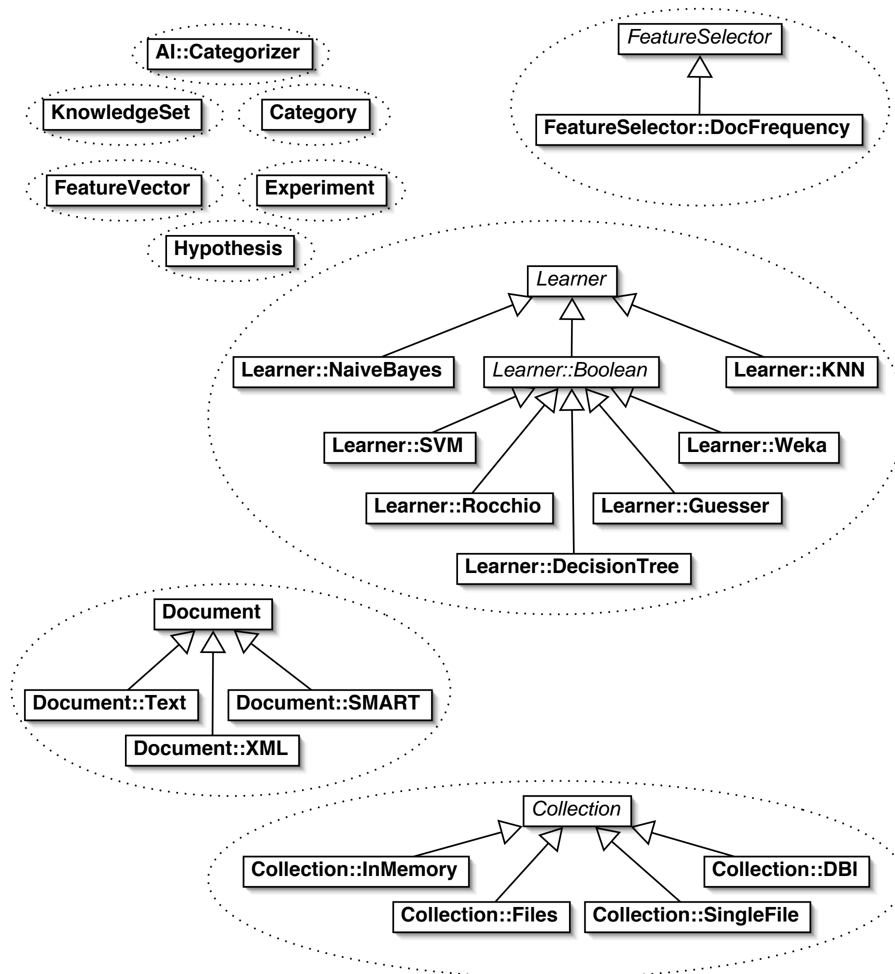

Figure 3.2: Inheritance diagram for `AI::Categorizer`

`egorizer` framework. Note that this diagram illustrates the *capabilities* of the framework more than it illustrates its *architecture*. For instance, the framework currently understands several document types, including plain text documents and documents stored in the format used by the SMART Information Retrieval system [4]. If the framework is extended by writing additional subclasses of existing classes, the capabilities increase without changing the basic architecture of the framework.

Note that the inheritance diagram is not particularly enlightening about how various classes cooperate to perform text categorization tasks. The inheritance relationships are set at compile-time and do not change while the framework is in use. Note also that in any given application, only one member of each inheritance hierarchy will typically be instantiated; an application using the SVM algorithm for categorization will not instantiate other Learner classes. So while the inheritance hierarchy diagram provides information about the capabilities of the framework, it provides little information about the structure of an application that uses the framework.

Another way to examine the framework is to examine the run-time relationships between its classes. This often provides a much more enlightening analysis of a framework, since modern framework design often favors object composition over class inheritance for its important structural relationships [18, p. 20].

The diagram in Figure 3.3 shows the most important run-time relationships between classes in the `AI::Categorizer` framework. In this diagram, no inheritance relationships are shown—any inheritance hierarchies are represented only by their parent classes. In general, a class and its subclass will share an interface and have identical relationships to other classes, but will differ in implementation. Therefore, the relationships indicated in this diagram indicate stable aspects of the framework that do not change when the framework is extended by subclassing.

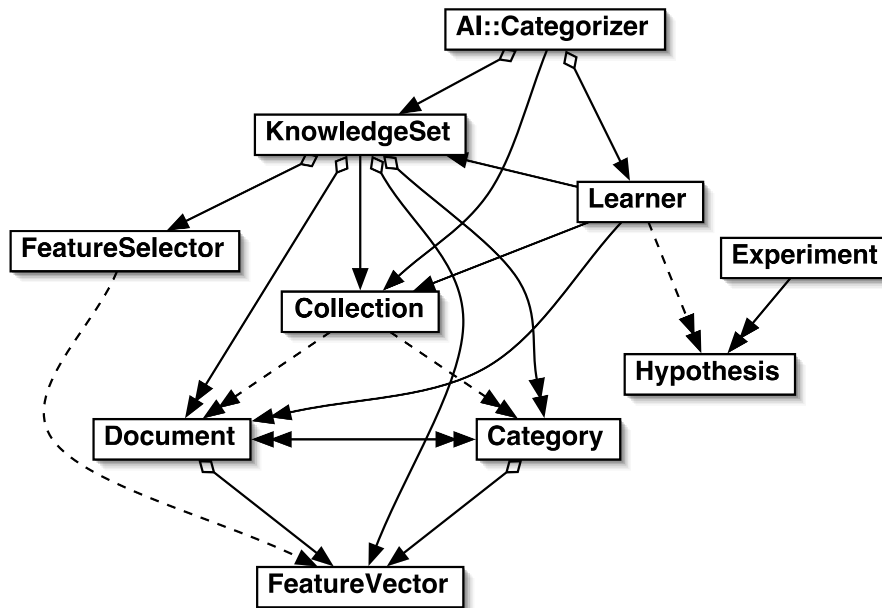

Figure 3.3: Class composition diagram for `AI::Categorizer`

3.4 Framework classes

Some examination of the basic relationships between classes and the responsibilities of each class is helpful before looking at the design in more detail. The following classes form the main framework roles in `AI::Categorizer`. For exposition purposes, the UML class specification boxes do not list every attribute and method of each class, rather only the most important ones for this discussion. In particular, all classes define `new()` constructor methods, not shown in the UML specifications, that accept various parameters (usually corresponding to the member attributes). For a complete reference, please see the `AI::Categorizer` documentation.

3.4.1 KnowledgeSet

KnowledgeSet
`Document[]` documents Category[]categories `FeatureSelector` feature_selector
`void` documents() `void` categories() `void` scan_features(`Collection`)

The `KnowledgeSet` class represents a set of processed documents, a set of categories, and a many-to-many mapping between the two sets. Processing may involve tokenization, stopword removal, linguistic stemming, feature selection, and vector weighting. Note that the term "knowledge set" is somewhat unique to this project, though the term "knowledge" is often used to describe an organization's collection of data used as the training set $\mathcal{T}^r$ when building a TC application.

A `KnowledgeSet` contains references to many `Document` objects and `Category` objects. It uses `Collection` objects to instantiate `Document` and `Category` objects. It uses a `FeatureSelector` object to perform feature selection. It also contains a `FeatureVector` object representing the features present in all documents.

3.4.2 FeatureSelector

FeatureSelector
`number` features_kept
`FeatureVector` reduce_features(`FeatureVector`) `FeatureVector` select_features(`KnowledgeSet`) `FeatureVector` *rank_features*(`KnowledgeSet`) `FeatureVector` *scan_features*(`Collection`)

Feature selection is performed by subclasses of the abstract `FeatureSelector` class. Each `KnowledgeSet` object contains a `FeatureSelector` object—the `KnowledgeSet` provides the information necessary to do feature selection, and the `FeatureSelector` performs the desired feature selection algorithm. A `fea-`

tures_kept parameter sets an attribute of the same name in the FeatureSelector class which controls how aggressively the features will be reduced.

The abstract rank_features() and scan_features() methods must be implemented by concrete subclasses of FeatureSelector. They provide two ways to create a ranked list of the features in a corpus. rank_features() will examine a KnowledgeSet object which has already been populated with document data, returning a FeatureVector object representing the relevance of each feature in the corpus according to the feature selection criterion. The scan_features() method has the same result, but operates on a Collection object so that the most relevant features can be determined without first having to read the entire collection into memory. A concrete FeatureSelector::DocFrequency subclass implements these methods, and may therefore be used in a blackbox manner for feature selection.

3.4.3 Collection

Collection
hash category_hash
int count_documents() Document *next*() void *rewind*()

Because data sets in text categorization may be very large, and because their documents may exist in several different underlying storage mechanisms (e.g. as files in a filesystem, sections of a larger XML file, or fields in a database), a Collection class provides an abstract interface to a set of stored documents together with a way to iterate through the set and return Document objects.

A concrete subclass of Collection must implement the next() and rewind() methods for the specific kind of iteration handled. next() should return the next document in the collection, and rewind() should reset the collection's iterator. This might mean re-executing a database query or moving to the beginning of a file or directory stream.

The default implementation of count_documents() simply calls next() un-

til all documents in the collection have been exhausted, counting how many documents are returned. This can typically be replaced by a more efficient concrete implementation that doesn't need to instantiate each document as an object.

The type of iteration to be performed, as well as the location of the document resources, will be specified by parameters to concrete subclasses. Because the locations may not have any uniform structure across subclasses (for instance, one subclass could use a path on the local filesystem, another could use a username, password, and query for a database, and another could use a Uniform Resource Locator for network collections), the determination of the exact parameters to specify is left to the subclass implementations.

A `category_hash` parameter lets the caller supply a hash relating document names to categories—if this is not provided, category information will be determined while iterating through the collection and processing the document data.

A `Collection` object may be used in several contexts within the framework. For instance, a `KnowledgeSet` instantiates its Document and `Category` objects through a `Collection` object. A `Learner` object may also mass-categorize the `Documents` in a `Collection` object.

3.4.4 Document

Document
`string` name
`Category[]` categories
`FeatureVector` features
`void` *parse*(`string`)
`void` parse_handle(`filehandle`)
`void` create_feature_vector()
`FeatureVector` features()

Each text document is represented by a `Document` object, or an object of one of its subclasses. Each document class contains methods for turning document data into a `FeatureVector`. Each document also has a method to report which

categories it belongs to.

In the standard methodology, the `parse()` or `parse_handle()` methods create plain-text data from the native format of the document, and the `create_feature_vector()` method creates a `FeatureVector` object (stored as a data member) from the text data. A default implementation of `parse()` is not supplied in the base class.

Note that the `Document` class is not purely an abstract class, because a `Document` object may be constructed and supplied directly with a `FeatureVector` object. Perl does not enforce the concept of abstract classes, so an unimplemented `parse()` method is not a problem in this case. This can make subclassing unnecessary for special-purpose types of "documents" like images or sequences of biological data.

3.4.5 Category

Category
`string` name `Document[]` documents
`Document[]` documents() `boolean` contains_document(`Document`) `void` FeatureVector(`features`)

Each category is represented by a `Category` object. Its main purpose is to keep track of which documents belong to it, though it also contains methods for examining statistical properties of an entire category.

Every category must have a `name` string, because `Collection`s will usually use the string to map between documents and categories. The string name is also shown to users when presenting categorization decisions.

The `features()` method returns a vector representing the sum of all vectors of documents that belong to the category. This may be used during a Machine Learning process. If the learner requires a different kind of aggregate representation of a `Category` than a simple vector sum, then the `documents()` method should be used to construct it manually.

3.4.6 Learner

Learner
`void` train(`KnowledgeSet`) `void` *create_model*(`KnowledgeSet`) `Hypothesis` categorize(`Document`) `float` *get_scores*(`Document`)

The abstract `Learner` class provides an interface to train on a set of pre-categorized documents using the `train()` method and subsequently categorize previously unseen `Document` objects using the `categorize()` method. Its concrete subclasses implement specific categorization algorithms like Naïve Bayes, SVM, Decision Tree, and so on.

The `create_model()` and `get_scores()` abstract methods need to be implemented in concrete `Learner` subclasses. They are called internally by `train()` and `categorize()`, respectively. The `get_scores()` method returns categorization information in terms of per-category scores and a membership threshold. `categorize()` translates this data into a `Hypothesis` object representing the categorization.

3.4.7 FeatureVector

FeatureVector
`hash` features
`float` euclidean_length() `void` scale(`float`) `FeatureVector` intersection(`FeatureVector`) `float` dot(`FeatureVector`)

As discussed in Section 2.2.6, most categorization algorithms don't deal directly with a document's data, they instead deal with a *vector representation* of a document's features. Most often, documents are represented using the "Bag of Words" model [38, p. 10], i.e. a non-ordered, weighted set of features. The `FeatureVector` class provides an interface to the operations one may perform on these vector representations, such as querying features' presence or absence in a document, finding sums of vectors, and so on.

The default implementation of the FeatureVector class stores its data in a Perl hash. See Section 4.3.1 for discussion of an alternate approach.

3.4.8 Hypothesis

Hypothesis
Category[] all_categories hash scores float threshold
Category[] categories() Category best_category() boolean in_category(Category)

The result of asking a Learner to categorize a previously unseen document is a Hypothesis object. It may be queried for information about which categories were assigned, which category was the single most appropriate category, what scores were assigned to each category, and so on.

In order to support this range of behaviors, the Learner is required to create the Hypothesis object by specifying an appropriateness score for each category and a threshold for category membership. Any category whose score is above the threshold is considered assigned by the system to the given document.

3.4.9 Experiment

Experiment
Category[] categories
void add_hypothesis(Hypothesis) float micro_F1() string stats_table()

The Experiment class can examine the results of many categorization decisions (i.e., many Hypothesis objects) and may be queried for aggregate information about the results. This is often used in order to determine the quality (as measured by precision, recall, error, etc.—see Section 2.4) of a Learner on a collection of test documents.

The Experiment class uses the external module Statistics::Contingency from CPAN to store and compute its results. The Statistics::Contingency

module was created for the `AI::Categorizer` project, but was split from the framework code because it is useful for projects not involving the rest of the framework. The framework's `Experiment` class is therefore quite small, inheriting most of its functionality from the more general-purpose module.

`Experiment` contains an `add_hypothesis()` method for adding `Hypothesis` objects to the set of data to be summarized, and implements a `stats_table()` method for showing a summary of the data in table format with a specified number of significant figures.

3.4.10 AI::Categorizer

AI::Categorizer
`KnowledgeSet` knowledge_set `Learner` learner `Collection` test_set
`void` scan_features() `void` read_training_set() `void` train() `void` evaluate_test_set() `void` run_experiment()

An umbrella class `AI::Categorizer` sits above the rest of the classes providing a convenient interface to a complete system for text categorization. Most applications built using the framework will instantiate an object of this class. Note that the term `AI::Categorizer` can refer either to the framework as a whole, or to the umbrella class. The distinction will be made clear in this text where it is necessary to do so.

The main benefits of having an umbrella class in the framework are that the object constructor mechanism described in Section 3.5.5 can operate consistently across the entire framework, and that it provides a very high-level interface, requiring very little application-specific code to invoke the framework's functionality.

The most important attributes of the `AI::Categorizer` class are a `Learner`, a `KnowledgeSet` that the `Learner` trains on, and a `Collection` of documents

that the Learner can be tested on. Because not every usage of this class need involve both training and testing, the KnowledgeSet and Collection attributes may be null if they are not needed by the particular application.

The scan_features() method invokes feature selection using the KnowledgeSet's FeatureSelector. The resultant list of desired features can be used by the read_training_set() method which populates the KnowledgeSet with data from the training corpus. The train() method invokes the Learner's train() method on the KnowledgeSet, and the evaluate_test_set() method invokes the Learner's categorize_collection() method on the test collection. A run_experiment() method automates the running of these four methods and shows the user a summary of the results.

3.5 Design Patterns in AI::Categorizer

The real power and intellectual content of any framework lies not in the design of its individual classes, but in the interfaces between the classes and the way objects collaborate to solve problems in the framework's application domain [15, p. 31]. These relationships can be quite complicated and difficult to explain, yet understanding them is essential to understanding the framework.

In this section, certain important local structures in the AI::Categorizer framework design will be discussed using the language of design patterns (see Section 1.2.1). The "Iterator," "Composite," "Adapter," "Strategy," and "Factory Method" patterns are discussed, and specific examples from AI::Categorizer show how they are applied within the framework. These are not by any means the only instances of common design patterns in the framework, nor do the specific patterns in [18] provide a complete catalog of all possible patterns in software. This discussion also does not give complete coverage to all design-related issues involved in AI::Categorizer. But patterns often provide a starting point for design discussion, and their use has been found beneficial in many diverse arenas [19], so they are used here in the hope that they clarify

the important design issues.

3.5.1 Iterator

The Iterator pattern provides "a way to access the elements of an aggregate object sequentially without exposing its underlying representation." [18, p. 257] Its main purpose is to decouple the traversal process on an object's aggregate members from the object's internal data structure implementation. In this way, clients can iterate through aggregate objects without knowing the objects' internal structure.

In the `AI::Categorizer` framework, it is often necessary to iterate through collections of documents and perform some action on them. For example, the documents may form a training set for a `Learner` to base a model on, or they may form a test set on which to evaluate the model.

The `Collection` class implements the Iterator pattern [18, p. 257] over documents in the framework. Figure 3.4 shows the main relationships involved in this pattern.

[18, p. 259] suggests that the most common reasons for using a formal custom iterator are:

- to access an aggregate object's contents without exposing its internal representation.
- to support multiple traversals of aggregate objects.
- to provide a uniform interface for traversing different aggregate structures.

The first and third reasons are most germane to the TC document iteration process. As explained in section 2.2.1, it is important that the framework can directly import documents from their various underlying storage mechanisms in order to prevent unnecessary duplication of data. In order to hide the details of the storage mechanism from the rest of the framework, a `Collection` object retrieves documents from the storage mechanism and returns them as `Document`

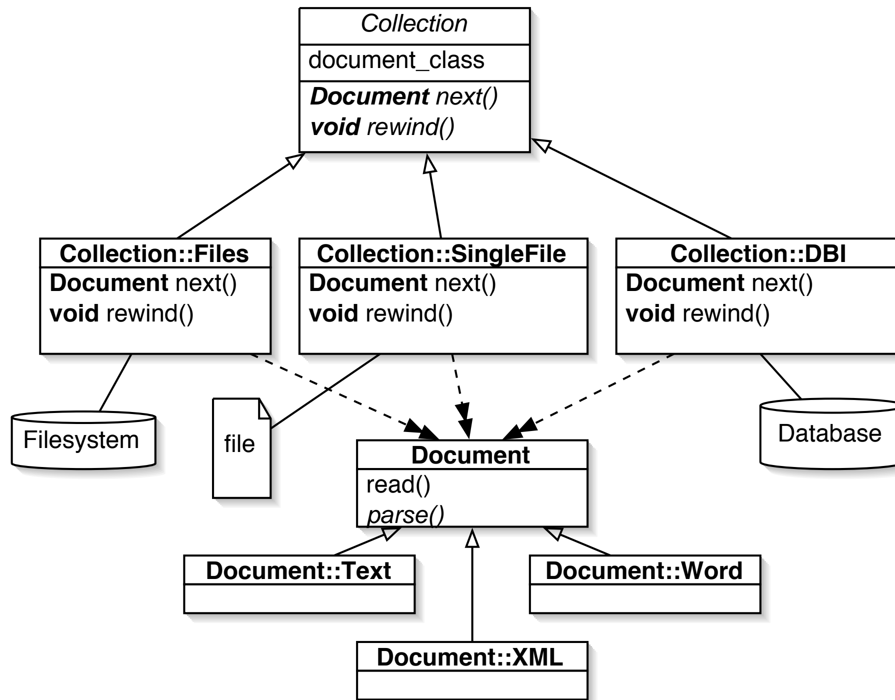

Figure 3.4: The Iterator pattern in the `Collection` class

objects. It provides a unified interface to iteration over stored documents so that the various classes that need to perform this iteration (chiefly `Learner` and `KnowledgeSet`) don't need to be aware of storage issues. In this sense, the "internal representation" of the aggregate structure is often external to the framework itself—it may be files in a filesystem, entries in a database, records in an XML document, or another mechanism.

In addition to providing a generic interface to a stored collection of documents, the Iterator pattern allows clients of the `Collection` class to use memory efficiently. A `Collection` object will typically defer creation of its `Document` objects until its client calls its `next()` method. In this way, the `Collection` doesn't store all the `Document` objects in memory simultaneously—if the client needs to do so, it can, or it can merely query properties of each document and dispose of them in turn.

Note that the `Collection` class defines a `next()` method, but no `previ-`

`ous()` method. This is largely because common document storage mechanisms like filesystems or databases typically only have one-directional iterators. Insisting that `Collection` classes needed to implement a `previous()` method to support bidirectional iteration would impose an unreasonable burden on them.

In order to decouple the storage mechanism from the internal format of documents (see section 2.2.2), `Collection` classes can cooperate with any subclass of the `Document` class. The client of the `Collection` class informs it that it should instantiate documents using a certain `Document` subclass. Since the `Document` subclasses share a common interface, `Collection` may remain ignorant of all internal document formatting issues, passing data to the proper constructors in order to instantiate `Document` objects.

3.5.2 Composite

The Composite pattern "lets clients treat individual objects and compositions of objects uniformly." [18, p. 163] It is often used to represent trees or other data structures in which the form of a subset of the structure is not qualitatively different from the form of the entire structure. In simple terms, this means that the same kinds of operations—iteration over sub-nodes, inspection of the root node, and so on—can be performed on the entire tree, a subtree, or even a single node.

In fact, the Composite pattern does not apply only to tree structures. It applies whenever a self-similarity exists between the whole and the parts in a part-whole hierarchy.

One instance of this kind of structure in Text Categorization is in so-called "ensemble learners," also known as "classifier committees." An ensemble learner is a categorizer that combines the results of a set of other categorizers in some way to arrive at a categorization result of its own [38, p. 30]. Often, an ensemble learner may outperform each of its constituent members on the general categorization task [39].

To implement ensemble learners within `AI::Categorizer`, the Composite

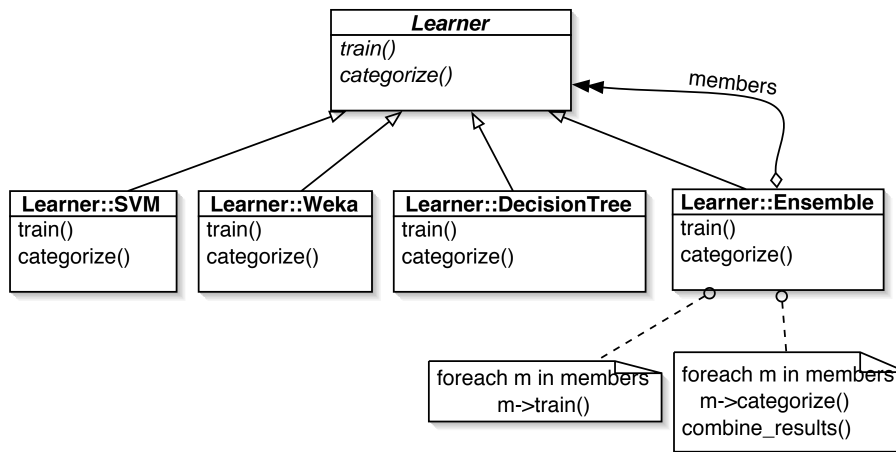

Figure 3.5: The Composite pattern in the `Learner::Ensemble` class

pattern may be applied to the `Learner` class to create a `Learner::Ensemble` subclass. Figure 3.5 shows the classes participating in this pattern.

Since `Learner::Ensemble` is a subclass of the abstract `Learner` class, it conforms to the `Learner` interface. This is crucial to implementation of the Composite pattern—it means that clients may use the `Learner::Ensemble` class without knowing that it implements an ensemble learner behind the scenes. In this way, transparent ensemble learning is achieved through polymorphism.

According to [38, p. 30], ensemble learning techniques can be specified by (1) a set of individual learners (the "members" in Figure 3.5), and (2) a mechanism for combining the output of the individual learners. The `Learner::Ensemble` class can provide generic support for creating the member learners of the ensemble, but the combination mechanism may take many different forms. Such algorithms are an active area of Machine Learning research. As such, `Learner::Ensemble` may be subclassed in order to implement different combination mechanisms. Since these subclasses implement the combination algorithm in different ways, they may themselves be seen as carrying out a Strategy pattern (see section 3.5.4).

3.5.3 Adapter

The Adapter pattern "converts the interface of a class into another interface clients expect." [18, p. 139] It is commonly used when an existing resource provides the functionality necessary for a certain task, but the interface of that resource doesn't match the interface necessary for the environment in which that task must be performed. For example, a framework may require that a particular role is implemented by subclasses of a certain abstract class. This helps unify functionality by taking advantage of polymorphic abstraction [15, p. 5]. That functionality may already be present in an existing body of code outside the framework. An Adapter can help bridge the gap between the two code bodies by letting the external code function inside the framework.[1]

Many developers in the text categorization community create their software as demonstrations of novel algorithms, or as stand-alone libraries that implement one small part of the overall text categorization task. The majority of cutting-edge research will be implemented in this way, if a public implementation is available at all. In order to leverage this work for a categorization framework, some adaptation is invariably necessary. Unless a developer happened to be using `AI::Categorizer` as a development environment, her implementation will not be directly usable as a framework element. Thus Adapters provide a mechanism for keeping the framework current with advances in the field of text categorization.

Figure 3.6 shows how the Adapter pattern is present in `AI::Categorizer`'s `Learner` class. The abstract `Learner` class specifies a common interface that all subclasses must conform to. Several of its concrete subclasses implement their functionality using a framework-external resource. For example, `Learner::DecisionTree` uses the stand-alone module `AI::DecisionTree` for implementation. `Learner::Weka` is a wrapper around the "Weka" Machine Learning system. `Learner::SVM` is a wrapper around a framework-external `Algorith-`

[1] An Adapter may sometimes be called a Wrapper. Both terms will be used in this discussion.

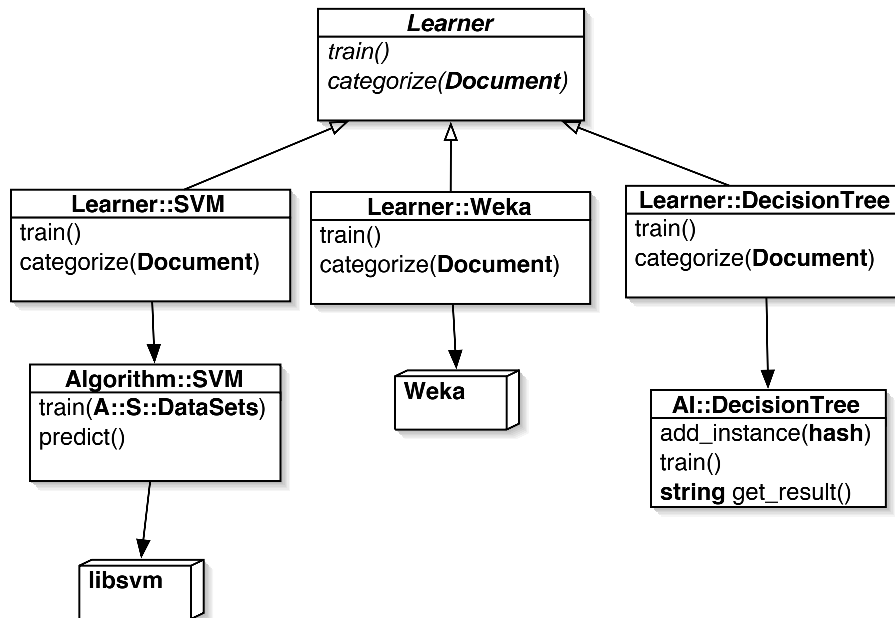

Figure 3.6: The Adapter pattern in the Learner class

`m::SVM` module, which is itself a wrapper around the `libsvm`[7] C library.[2]

Note that these four Adapter examples exhibit three very different applications of the Adapter pattern. `Learner::DecisionTree` exhibits a very straightforward Adapter usage as presented in [18]—an existing stand-alone class exists that implements the needed functionality, and its interface is adapted to the framework requirements by a simple wrapping subclass. The `Learner::SVM` wrapper is also fairly straightforward. The other two wrappers, however, reflect the highly heterogeneous nature of the text categorization domain. The main reason for the adaptation in the `Algorithm::SVM` class is to provide a Perl interface to a C library. The `Learner::Weka` adapter combines language adaptation (in this case, Java to Perl) with functionality transformation (mapping Weka's methods to the required `Learner` interface).

Adapters can create design flexibility. The current implementation of `Learner::Weka` interfaces with Weka through its command-line interface, but this is

[2]Note that `Learner::SVM`, Weka, and `libsvm` are not part of the contributed work of this thesis, as they are written by other authors.

not a design constraint. Future implementations may embed the Weka system inside the `Learner::Weka` module for reasons of efficiency or platform compatibility. Because this interface is hidden using an Adapter pattern, the implementation may be changed freely.

The differences in interfaces between the Adapter and the Adaptee may be merely historical, or they may reflect different needs in different domains. The Adapter must conform to the interface of its abstract superclass, which is typically designed to be independent of subclass abilities and implementations. The Adaptee may be designed for use in a different arena, with extra functionality or an interface that takes full advantage of its capabilities.

Using Adapters may bring major benefits in the area of reusability. Obviously, classes won't have to be re-implemented if the functionality can be adapted from an existing implementation. Second, and perhaps more importantly, classes initially implemented for a framework may be converted into Adaptee classes that are usable in isolation. This can bring them better exposure in other projects and thus more feedback, maturing them quickly. This can be a major win, because iteration is considered a limiting factor in framework development [15, p. 75], so any process that speeds up maturity in framework components can have a large impact. Adapters can also force a more robust encapsulation of design in the Adaptee, bringing benefits in the conceptual and technical segmentation of the framework.

3.5.4 Strategy

The Strategy pattern defines "a family of algorithms, encapsulates each one, and makes them interchangeable" [18, p. 315]. It is used when a domain task needs to be carried out, but there may be several ways to carry out that task, and it is important to let the user or client choose from among these alternatives.

An important concept in the Strategy pattern is that of "behavior." In [18], the Strategy pattern is recommended when "many related classes differ only in their behavior." In this context, a distinction is made between an algorithm's

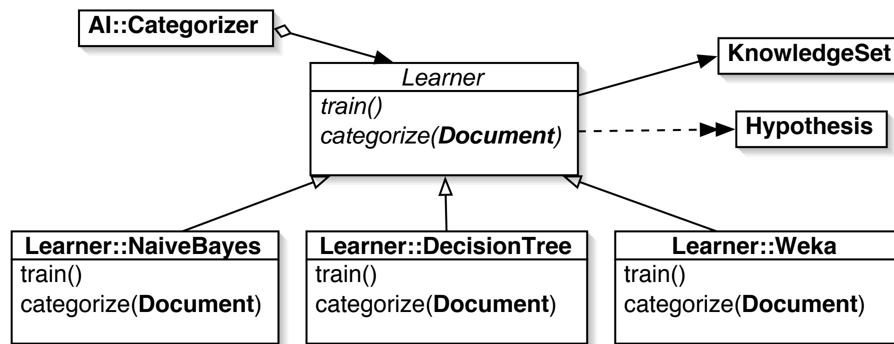

Figure 3.7: The Strategy pattern in the Learner class

purpose and its behavior. For example, a set of algorithms for finding line-break points in text paragraphs have a common purpose (to accomplish the line-breaking task), but they may carry out their task in different ways. The algorithms may make different trade-offs in terms of speed and memory, or they may try to optimize different aspects of the task. Since it is impossible to satisfy all clients in all situations with a single choice of algorithm, it is desirable to encapsulate each algorithm in a class that can be chosen or extended by the client.

The field of text categorization has several natural applications for the Strategy pattern. One of the primary concerns of most TC researchers is the development of novel algorithms for various aspects of the categorization task, so it is essential for these algorithms to be easy to vary in a categorization framework. In the language of [15], these algorithms are framework "hot spots."

The most obvious Strategy application in `AI::Categorizer` is the `Learner` class and its subclasses. These classes all have a common task to perform: that of training a categorizer and categorizing unseen documents. The various subclasses represent very different ways to accomplish that task. Importantly, the results of the task, and not just the internal mechanism that performs it, may be different depending on which `Learner` subclass is used.

Figure 3.7 shows how the Strategy pattern appears in the `Learner` class and its subclasses. Three concrete subclasses are shown that implement specific Ma-

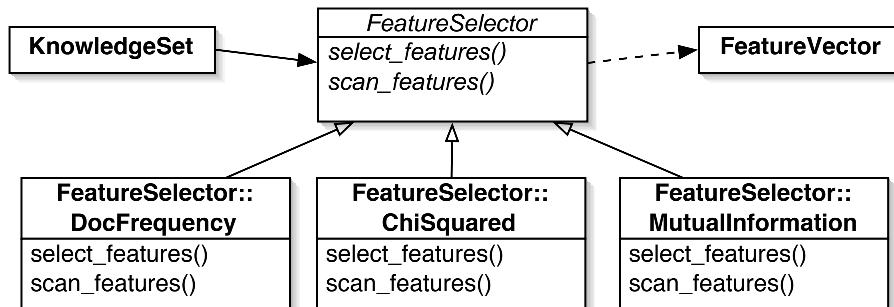

Figure 3.8: The Strategy pattern in the FeatureSelector class

chine Learning algorithms (see Figure 3.2 for other `Learner` subclasses currently implemented). From the point of view of the client `AI::Categorizer` object, all `Learner`s have the same interface and may therefore be treated uniformly. The framework user or application designer, however, may choose judiciously among subclasses depending on the particular needs of the application. Customizability of the Machine Learning algorithm is of paramount importance to the framework since it would be useless to most researchers if this were not the case. Many researchers may wish to write their own `Learner` subclasses using this portion of the framework in the "whitebox" paradigm. Other researchers, and most application developers, will want to use existing framework classes in a "blackbox" framework usage style [15, p. 10]. Either method is supported.

Each `Learner` subclass must implement the abstract `train()` and `categorize()` methods in order to perform the two essential tasks of any `Learner`. The `train()` method examines a `KnowledgeSet` object and builds an internal (and opaque) model that will be used to categorize future documents. The `categorize()` method takes a `Document` object as an argument and returns a `Hypothesis` object representing the outcome of categorization based on the model.

Another application of the Strategy pattern is shown in Figure 3.8. Here, the varying algorithm performs feature selection, another framework hot spot. There has been much activity in current research on improving feature selection for different scenarios [49, 51], so customization in this area is also essential.

To perform feature selection, a KnowledgeSet object invokes either the select_features() or scan_features() method of the FeatureSelector object, depending on whether a complete KnowledgeSet or a Collection object should be examined. Examining a Collection iteratively requires less memory because the documents don't have to be loaded into memory all at once, but it requires a separate pass through the data. The choice of which method to run is made in response to user specification.

Because select_features() and scan_features() are virtual methods in the parent class, any concrete subclass must implement these methods according to the particular algorithm the subclass represents. As of this writing, only the FeatureSelector::DocFrequency subclass is implemented, but the other subclasses in the diagram are planned.

3.5.5 Factory Method

In any framework of sufficient size and customizability, attention must be paid to the issue of how specific classes are chosen for the various framework roles, how these classes are instantiated, and how the instantiated objects are connected to each other. In the simplest possible case for the framework developer, the framework client code must create all objects and manually connect them to each other. For instance, in AI::Categorizer, the client code might create a KnowledgeSet object, a Learner object, an AI::Categorizer object, a Collection object, then populate the AI::Categorizer object with KnowledgeSet and Learner objects, and the KnowledgeSet with a Collection, thus satisfying the structural relationships indicated in Figure 3.3 on page 38. An approach like this is diagrammed in Figure 3.9.

This approach works, but it is error-prone and cumbersome. It forces every client to specify the framework relationships explicitly, when in fact these are fundamental relationships of the *framework*, not of the client code. It makes little sense for this structural code to be outside the framework and even less sense for it to be duplicated in every application that uses the framework. Note

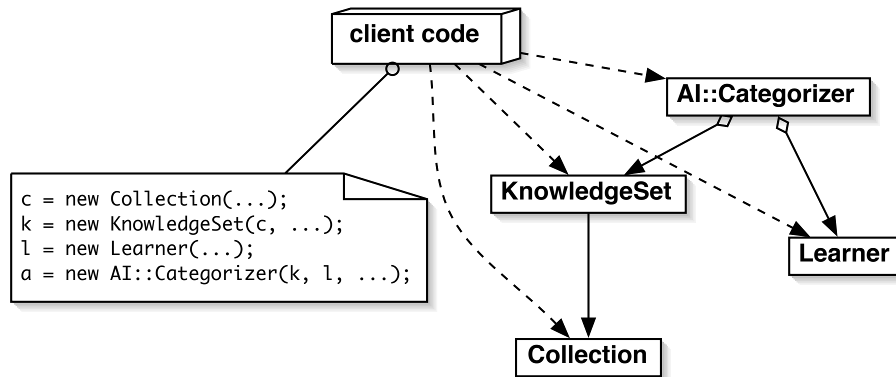

Figure 3.9: A client-side approach to object construction

too that Figure 3.9 only shows a small part of the framework being used—in reality, the client code would have to accept responsibility for creating all the objects in the framework, not just the four pictured here.

For these reasons, it is often better if the framework can provide support for object creation and enforcement of the framework relationships. An example of this situation is pictured in Figure 3.10. Here, the patterns of object creation more closely follow the class relationships that will be used at runtime. This design is moving closer toward a factory-style pattern in which object creation is delegated to another object [18]. defines two specific kinds of factory patterns: “Factory Methods” and “Abstract Factories.” Figure 3.10 does not fit either of these patterns exactly, but it does fall under the general category of factory object creation.

A scheme like that in Figure 3.10 has both advantages and disadvantages compared with that in Figure 3.9. One obvious advantage is that the client code is greatly simplified, because it needn’t create any framework objects except the top-level object, and because it doesn’t have to link the objects to each other. This eliminates redundancy in multiple client code bases and allows the framework designer greater flexibility in redesigning the framework hierarchy. Another advantage is that the framework objects are created by the objects that use them, so class code can accept responsibility for its subordinate objects’

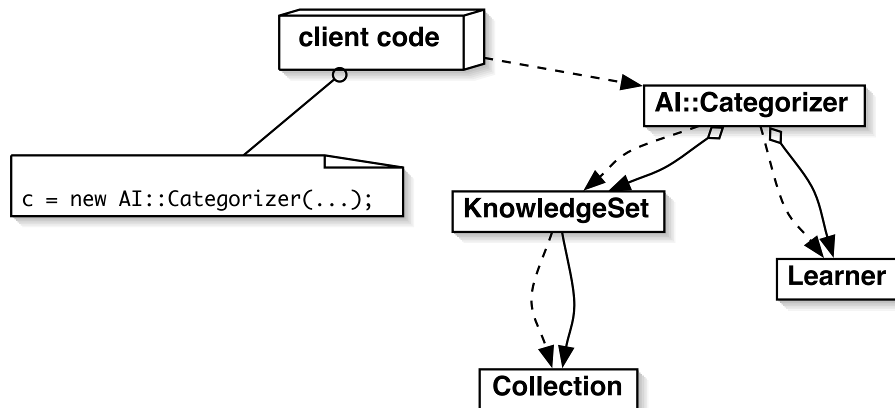

Figure 3.10: A framework-side approach to object construction

entire life cycles.

However, these properties can also be seen as disadvantages. If each framework object assumes all responsibility for creating its subordinate objects, then the client may not be able to control the creation process effectively. This is a problem for at least two important reasons. First, the client may wish to change some properties of the objects it creates. If it passes all constructor parameters to the top-level class constructor, then this constructor must have knowledge of all of its subordinate classes' parameters in order to affect their construction correctly. This would couple the framework classes too closely. Second, the client may (and frequently will) change which classes are participating in the framework hierarchy. If the `KnowledgeSet` class always creates a certain class of `Collection` object, then in order to substitute a different `Collection` class, the `KnowledgeSet` class would need to be subclassed—this means the top-level `AI::Categorizer` class would also need to be subclassed in order to create the new type of `KnowledgeSet`, leading to a proliferation of subclasses just to manage object creation. Clearly a better solution is needed.

In order to create a proper solution, some analysis of the problem is warranted. Part of the reason these creational issues are difficult is that no standard method exists to translate the framework's design relationships into code. Common programming languages have no built-in support for managing the

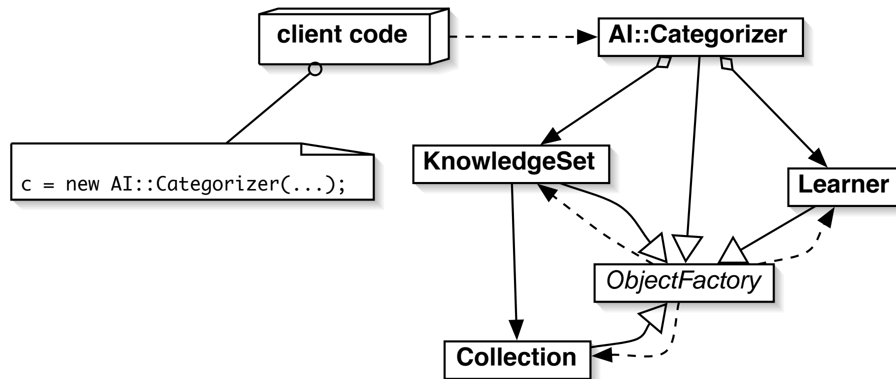

Figure 3.11: A centralized approach to object construction

patterns of creation necessary in frameworks. Contrast this with inheritance relationships, which are directly supported by object-oriented languages. For instance, a C++ or Java `class` declaration lists its superclasses explicitly, and Perl specifies inheritance via each class's `@ISA` array. Because inheritance is directly implemented by the language, it is easy for framework users to understand inheritance relationships, and these relationships are expressed straightforwardly in the framework code. For support of this point, consider object-oriented programming in languages like C that don't have inherent OO support. Understanding the inheritance structures can be much more challenging in this situation [15, p. 7].

With this perspective in mind, one solution is to create a way for each class to explicitly declare its constructor parameters and its relationships to other classes, and then let the framework manage object creation in a consistent, centralized manner based on these declarations. In a sense, this approach extends the implementation language to be able to express the important framework relationships directly rather than letting them emerge implicitly from patterns of usage in the code. Client code then supplies parameters that inform the top-level object about which classes should be instantiated and what parameters should be passed to each class's constructors, and the framework itself directs the object creation process.

Because many users of an application framework will be hesitant to depend on a modified, nonstandard version of the implementation language, `AI::Categorizer` uses inheritance to add these explicit declaration capabilities to every class participating in the framework hierarchy. Figure 3.11 shows an example of how this inheritance functions. The abstract *ObjectFactory* class[3] adds the ability for any class derived from it to declare the relationships discussed in the previous paragraph. It also manages the creation of subordinate objects. For instance, the top-level `AI::Categorizer` class declares that it contains both a `KnowledgeSet` and `Learner` object in an aggregation relationship. When an `AI::Categorizer` object is created, `KnowledgeSet` and `Learner` objects will automatically be created by the *ObjectFactory* according to the client code's parameters. The `KnowledgeSet` also declares that it will need to create `Collection` objects on demand, and calls creational methods provided by its *ObjectFactory* superclass when it needs to create them.

It is important to note that this is not a direct application of either the Factory Method or Abstract Factory patterns in [18]. The standard Factory Method pattern requires separate subclasses to create the concrete subclasses. A closer variation is the "Parameterized Factory Method" [18, p. 110], which lets the specific subordinate class be determined by switching among several known classes. This is closer to the data-driven approach employed in `AI::Categorizer`, but it doesn't address the issue of how the subordinate classes must actually be created at runtime. The Abstract Factory pattern is also similar in that the creation of multiple objects is centralized, but in `AI::Categorizer` a separate factory object is not necessary.

This approach effectively solves the problems with the first two approaches considered here. The client code is freed from having to create multiple framework objects, and the framework relationships are expressed explicitly in the framework code, not in the client code or implicitly in the framework implementation. Clients are also able to easily change which classes participate in

[3]The *ObjectFactory* name is used here only for discussion purposes. See Section 4.2 for the actual implementation details.

```
use AI::Categorizer;
my $c = new AI::Categorizer(...parameters...);
$c->run_experiment;
```

Figure 3.12: Highest-level interface to `AI::Categorizer`

the framework hierarchy and can specify constructor parameters without invoking the constructors directly. Framework code doesn't create subordinate objects directly, but defers creation to factory methods inherited from superclasses. In this way, subclassing is kept to a minimum, and the framework runtime structure can be highly parameterized.

3.6 Examples

Effective documentation is essential for the use and dissemination of any framework [15, ch. 21]. The `AI::Categorizer` distribution contains complete documentation of the user-visible classes. That documentation will not be reproduced here. Of use to the present discussion, however, are some simple examples of using the framework. Example code often forms one of the most important kinds of framework documentation since it shows concrete examples of framework usage [15, p. 498].

Figure 3.12 shows the highest-level interface usage, in which an entire experiment—training on a training corpus, testing on a test set, and showing results to the user—is performed by setting appropriate parameters in the constructor of the highest-level `AI::Categorizer` object. These parameters may include `learner_class` for specifying the class of machine learner that should be used, `stopword_file` for specifying a file containing a list of stop words, or `stemming` to indicate what type of linguistic stemming, if any, should be performed on the document data. The generic Factory Method mechanism described in Section 3.5.5 ensures that each parameter becomes an argument to the appropriate object constructor method.

```
use AI::Categorizer;
my $c = new AI::Categorizer(...parameters...);

# Run the separate parts of $c->run_experiment
$c->scan_features;
$c->read_training_set;
$c->train;
$c->evaluate_test_set;
print $c->stats_table;
```

Figure 3.13: Separate invocations of experimental phases

```
my $l = AI::Categorizer::Learner->restore_state(...);
while (...) {
  my $d = ...create a document...
  my $h = $l->categorize($d);
  print "Best category: ", $h->best_category, "\n";
}
```

Figure 3.14: Using `AI::Categorizer` for direct categorization of documents

Figure 3.13 shows a slightly lower-level interface to the framework. Here, the individual stages of the `run_experiment()` method from Figure 3.12 are run separately, and may in fact be run in separate programs on different machines if the `progress_file` parameter is used in order to save state between the stages.

In an applied setting, the application developer may need much finer control over the object behavior. For instance, the developer may not be very interested in the overall performance on a test set, but rather in the specific decisions of the trained categorizer on documents presented to it by users. Figure 3.14 demonstrates one simple such application, in which the trained categorizer is loaded into memory using the `restore_state()` method (see Section 4.3.3), then repeatedly asked to categorize documents using the `categorize()` method. Here the application uses the `best_category()` method to select only the single category with the best score, but another application may require different information from the `Hypothesis` object `$h`.

3.7 Limitations

In any software design process, choices must be made that determine the scope and direction of the project. In designing `AI::Categorizer`, these choices have been made in a way that tries to maximize the usefulness for the intended audience, the reuse of framework components, the framework's efficiency and flexibility, and the rapid development of applications. In some cases, these decisions may limit the capabilities of the framework. This section describes some of these limitations, explains the reasons for them, and proposes alternative ways to deal with the problems they present.

3.7.1 Structured Feature Vectors

The basic data model representing documents in the `AI::Categorizer` framework is the feature vector. In this model, certain features of each document (typically counts of words or word stems) are measured, and their values are represented as vectors in a vector space encompassing all document vectors in the training set. Each document vector is *flat*, i.e. an n-dimensional vector with no internal structure, where n is the total number of features in all the training documents. This representation has been shown to be very effective for Text Categorization applications [38, p. 10] and is crucial for such common TC algorithms as k-Nearest-Neighbor and Support Vector Machines.

However, many environments routinely use richer data models for documents. For instance, researchers in the Linguistics community often represent documents as hierarchical data structures indicating each syntactical element's relationship to the other syntactical elements in the document [26, ch. 11 & 12] [34]. Additionally, many structured HTML and XML business documents are represented using the Document Object Model, which provides a common programmatic interface to the logical structure of documents [48].

Because few TC techniques in the common literature take advantage of document structure, and because several techniques depend on unstructured vector

representations, the FeatureVector class only provides an interface to unstructured feature vectors. This class does leave the *implementation* of the vectors unspecified, however, so that different internal representations are possible (see Section 4.3.1 for more on this topic).

In an application using structured documents, two options exist for taking advantage of this structure using AI::Categorizer. One option is to "flatten" the structure of the document into a traditional feature vector representation. Unfortunately, it is not always clear to the developer how this flattening should be done, and the flattening process may lose valuable information about document structure, making categorization results sub-optimal.

Another option if the document structure is just a sequence of document sections and not arbitrarily nested structures is to use the AI::Categorizer framework's content_weights parameter. This allows each document to be divided into an arbitrary number of sections such as title, abstract, body, and so on, assigning "importance" weights to each section. These weights will be used when creating a feature vector from the document content, in effect automatically flattening the document into a traditional feature vector.

Neither of these two solutions allow the framework to truly deal with arbitrarily structured documents in any natural way. It is therefore to be understood that the framework is not currently capable of exploiting this structure very deeply, and this is a possible area of future work.

3.7.2 Hierarchical Categorization

Hierarchical categorization is the process of categorizing documents into a set of categories possessing a treelike structure. The hierarchical nature of the category set may be exploited for both increased efficiency and improved accuracy [13]. Because some common categorization problems are inherently hierarchical, the field of hierarchical categorization has seen significant attention in the research literature [38, p. 7].

In the AI::Categorizer framework, hierarchical categorization has not ex-

plicitly been supported in the architecture. The set of categories in any framework categorization task is assumed to be a simple list of named sets of documents with no hierarchical structure. However, there are at least two ways of dealing with hierarchical categorization tasks using the framework.

The first way is to simply transform the hierarchical set of categories into a simple flat list, by prepending each category's name with the names of all its parent categories. In this way, the framework will assign any category in the flat list of categories, and then the results can be transformed back into members of the hierarchical category set. The main advantage of this technique is that it is simple to apply, with a natural and transparent transformation between structured and flat category sets. The main disadvantage is that the system is not really performing hierarchical categorization at all, so it is not taking advantage of any of the hierarchical category structure for efficiency or accuracy improvements.

The second way to achieve hierarchical categorization using `AI::Categorizer` is to manually break the categorization task into several smaller tasks, building a separate machine learner for each splitting node in the category hierarchy. This is a common approach to hierarchical categorization in the literature [13, 24, 6], and it seems to naturally map the hierarchical problem into a hierarchical solution. The main disadvantage with this method is that the framework provides no direct support for creating a hierarchy of categorizers, so the client must create and maintain code for the hierarchical aspect of the task. This is another possible area of future work, and a fellow student in the Web Engineering Group at Sydney University is currently working toward a solution for using `AI::Categorizer` in hierarchical categorization.

Chapter 4

Implementation

With the functionality requirements and class designs in Chapter 3 as a guide, the `AI::Categorizer` framework has been implemented and released under an open source license [32, 11] as a part of this thesis and as a continuing project in Text Categorization. This chapter describes some of the implementation decisions that have been made in `AI::Categorizer` and provides some of the reasoning behind them.

4.1 Implementation Language

In order to provide maximum support for the kinds of real-world scenarios described in Section 3.2, it was determined that a broad-coverage, widely-used programming language should be used to implement the `AI::Categorizer` framework. Three extremely common object-oriented languages fulfilling these criteria are `C` (or rather its object-oriented derivatives like `C++` and Objective-`C`), Java, and Perl. Each of these languages has its advantages and disadvantages, and a full comparison between them is beyond the scope of this thesis. Perl was ultimately chosen for the `AI::Categorizer` project, providing the following benefits:

- Perl is widely known to be a powerful text processing tool [17, 29] [26,

p. 121], hence it should be relatively easy for users of the framework to customize its processing capabilities.

- A large number of contributed Perl modules are freely available for many different tasks on the CPAN [42], extending the domain of applicability of the framework.

- Perl is an expressive high-level language that allows for rapid prototyping, so the framework developers and application developers can experiment with several alternative designs fairly quickly.

- Perl is widely deployed and is part of all standard Unix distributions. It is available for most platforms that have a `C` compiler, and because of common high-level interfaces, Perl code written on one platform is often more portable to other platforms than the equivalent `C` code would be.

- Perl can be embedded within applications written in other languages, particularly in `C/C++` applications using Perl's embedding interface, or in Java applications using the JPL toolkit. This allows for maximum reusability of the framework as described in Section 3.2.2.

- Code from other languages can be embedded within Perl applications using either the XS extension mechanism for `C` code, or the Inline embedding mechanism for several languages, including `C` and its derivatives, Java, Tcl, Assembler, and Python, among others. This allows the framework to use efficient data structures and algorithms implemented in other languages if necessary, while keeping the convenient high-level interface in Perl. It also allows integration with existing code libraries from various sources without locking the developer into a language choice.

- There is an active community of users interested in using a Perl-native text categorization framework. Several community members have already contributed feedback, bug fixes, and application ideas for the `AI::Categorizer` project.

```
package AI::Categorizer::Learner;
use base 'Class::Container';
use Params::Validate qw(:types);

__PACKAGE__->valid_params
  (
   knowledge_set  => { isa => 'AI::Categorizer::KnowledgeSet',
                       optional => 1 },
   verbose        => { type => SCALAR,
                       default => 0 },
  );

__PACKAGE__->contained_objects
  (
   hypothesis => { class => 'AI::Categorizer::Hypothesis',
                   delayed => 1 },
   experiment => { class => 'AI::Categorizer::Experiment',
                   delayed => 1 },
  );
```

Figure 4.1: An example of `Class::Container` usage from the `Learner` class

4.2 Framework constructor methods

To implement the behavior discussed in Sections 2.2.8 and 3.5.5, the `Class::Container` module from CPAN[1] implements the abstract parent class ***`ObjectFactory`*** shown in Figure 3.11. It provides the generic specification of object constructor parameters as well as generic mechanisms for creating subordinate objects within the framework.

Figure 4.1 shows a simplified example of `Class::Container` usage in the `Learner` class from `AI::Categorizer`. The `valid_params()` and `contained_objects()` class methods are inherited from the superclass `Class::Container`, and they provide the mechanism by which each class can declare its main constructor interface. In this case, the `Learner` class declares that it accepts two constructor parameters, called `knowledge_set` (a `KnowledgeSet` object which will form the training set for the learner) and `verbose` (an integer specifying

[1]The `Class::Container` module was written by Ken Williams for a previous project [33] and greatly extended for the `AI::Categorizer` project.

the amount of status information to show the user during the training process).

The `contained_objects()` method lets the `Learner` class declare its subordinate objects in the framework architecture. In this case, each `Learner` will create `Hypothesis` and `Experiment` objects at runtime; `Hypothesis` objects are created in the `categorize()` method when any document is categorized, and `Experiment` objects are created in the `categorize_collection()` method to organize and report the results of categorizing many documents. The `Learner` will create these objects on demand using `create_delayed_object()`, also inherited from `Class::Container`, as a factory method. In the object specification, the "delayed" flag indicates that the objects will be created on demand in this manner—if this flag were not specified, the objects would be automatically created during the `new()` method in an aggregation manner [18, p. 22].

4.3 Data Structures

To a large extent, the data structures used in `AI::Categorizer` are unspecified, since the framework specification dictates only the framework interface methods and the relationships among classes. However, the concrete implementations of several classes must choose implementation details, and those details are described here.

4.3.1 Feature Vectors

Many parts of the framework code must manipulate feature vectors, which we define as a set of key-value pairs relating document features (which may be words, word stems, 2-word combinations [bigrams], or other derivatives of document data) to values (which may be frequency counts or other weighted measures of importance). Some examples of this include the features of an individual document, the aggregated features of a knowledge set or of the documents belonging to a particular category, or a vector of features whose weights have been assigned by a particular `Learner` implementation.

The `FeatureVector` class provides both a concrete implementation of a feature vector interface and a base class from which other classes may inherit if they wish to use a different internal representation of the data or extend the capabilities of the base class. Perl provides hash tables (sometimes called "dictionaries" in some languages) as a language-level data structure, and the default implementation in `FeatureVector` uses these as its mapping between features and values. This provides the following benefits:

- Insertions, deletions, and lookups are all $O[1]$ operations, so the size of the feature set can grow without any penalty on the time to perform these operations.

- Hashes can store *sparse* information efficiently, meaning only nonzero entries in a vector need to be stored. This can be important if the dimensionality of the ambient vector space is very large, because memory savings of 1-3 orders of magnitude may be realized.

- The hash data structure stores the key as a string. This may be the word or word stem from the document itself, avoiding the need to use a separate lookup table to translate from the actual document features to the keys of feature vectors.

However, there are certain liabilities with this approach as well:

- Perl's implementation of most data structures is fairly memory-greedy in order to provide benefits like automatic memory allocation and transparent casting. This can cause low-level data structures to consume much more memory than needed, as is the case for the base `FeatureVector` class.

- The hash key in the feature vector is always stored as a string, but it would be more compact to store it as a single integer representing an index into a global array of all features in the knowledge set, and store the global array separately.

To address these liabilities, a developer might prefer a feature vector implementation that stores its data as integer/float arrays in `C`-level data structures in the manner of [30]. This would be most useful when working with a large corpus or when using a system with a relatively small amount of physical memory. The drawbacks with using this approach would be that a separate structure mapping document features to integers would need to be maintained, and that searches through a feature vector for a specific feature would become an $O[\log(n)]$ operation, where n is the number of nonzero entries in the feature vector. In practice, the latter issue is usually not important, because each document vector is fairly small, and the difference between $O[1]$ and $O[\log(n)]$ may be insignificant compared to the constant overhead costs of the operations.

A `FeatureVector` subclass implementing the structure described in the previous paragraph is currently under development by another researcher at the University of Sydney, though it is not yet a part of the `AI::Categorizer` framework. Another `FeatureVector` subclass (`FeatureVector::FastDot`) which uses a greatly simplified version of the same structure, optimized for repeated dot-product calculations but otherwise identical to the standard `FeatureVector` class, has been completed. However, because it will be rendered largely obsolete by the other project underway, it will probably not become a part of the framework distribution.

4.3.2 Sets of Documents or Categories

In several places in the `AI::Categorizer` code, sets of Document or Category objects need to be created and manipulated. This needs to be done in such a way that insertion, deletion, iteration, and retrieval are all very fast operations, because these operations will be fundamental to most `Learners`' training methods.

To fulfill the above requirements, a Perl hash structure is used to store sets of objects, and this structure is encapsulated in the `ObjectSet` class. This class imposes two restrictions on its usage. First, because the keys in Perl

hashes must be strings, each object stored in an `ObjectSet` must be identified by a string, which must be given by the value of the object's `name()` method. Second, because hashes store their elements in an order that makes each of the four above-mentioned operations `O[1]` operations, any inherent ordering of the Document or Category objects is lost.

4.3.3 Saving state

In order to store the state of a trained categorizer, of a `KnowledgeSet` containing a training corpus, or of any other important object in the `AI::Categorizer` framework, a generic interface has been created for the serializing of objects to disk and the subsequent restoring of the serialized structure back into an object in memory. Two object methods, `save_state()` and `restore_state()`, are defined in the `Storable` class[2], from which the other framework classes inherit.

The default implementation of `save_state()` merely traverses the given object's internal data structure, storing the object's Perl-level structure as a file in a directory. The directory path is specified by the caller.

Classes that use non-Perl data structures (for instance, classes like `Learner::SVM` or `Learner::DecisionTree` that use structures implemented in external `C` code) may override the default `save_state()` method in order to invoke alternative serialization mechanisms.

[2]This discussion refers to the `AI::Categorizer::Storable` module, not the `Storable` module available on CPAN. In fact, the `Storable` CPAN module is used internally by `AI::Categorizer::Storable` to perform the data serialization, but this is not visible to the developer.

Chapter 5

Evaluation

In order to evaluate the quality of the `AI::Categorizer` framework, several aspects of the framework have been tested. The three main areas tested are quality of categorization, efficiency, and ease of use. For testing the quality of categorization and efficiency, performance is measured on categorization tasks using several different data sets.

Section 5.1 describes the data sets used during testing. Section 5.2 presents various measurements of how accurately the framework performs on these data sets, and Section 5.3 discusses the computational efficiency of the framework. Section 5.4 discusses the ease of use of the framework in different contexts.

5.1 Corpora

During development and testing, several data sets, or "corpora," were used for framework testing and application building. Since the framework will behave differently on different data sets, it is important to understand the characteristics of each corpus. For instance, different feature selection and categorization algorithms may scale differently in relation to the size of the training corpus, both in terms of efficiency and accuracy [6]. Also, the specifics of the categorization problem in each corpus may be more amenable to one categorization

technique or another.

The corpora used for evaluation are listed in this section. Each corpus consists of a set of documents $\mathcal{D}$, a set of categories $\mathcal{C}$ that documents may be assigned to, and a domain-expert-made choice of category assignments for each document. These category assignments are considered to be completely correct, and they form a standard against which the TC system's categorization on the test documents can be measured. For each corpus, the set $\mathcal{D}$ is divided into a training set $\mathcal{T}^r$ and a test set $\mathcal{T}^e$.

Unless otherwise noted below, a list of common English words from [35] was used as a "stoplist," or a set of terms to completely ignore when processing documents. This is a common technique from Information Retrieval [26, sec. 15.1.1], as it is assumed that these words possess little or no information about the target categories, and that they will only slow processing and add noise to the data.

5.1.1 ApteMod

The ApteMod version of the Reuters-21578 corpus has become a standard benchmark corpus in evaluating Text Categorization systems [50, 21]. In terms of evaluating the `AI::Categorizer` framework, it provides an opportunity to compare the performance of `AI::Categorizer` with other implementations of the same TC algorithms so that the correctness of the present implementation can be verified.

ApteMod is a collection of 10,788 documents from the Reuters financial newswire service, partitioned into a training set with 7769 documents and a test set with 3019 documents. The total size of the corpus is about 43 MB. It is available for download from `http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.html`.

The distribution of categories in the ApteMod corpus is highly skewed, with 36.7% of the documents in the most common category, and only 0.0185% (2 documents) in each of the five least common categories. In fact, the original

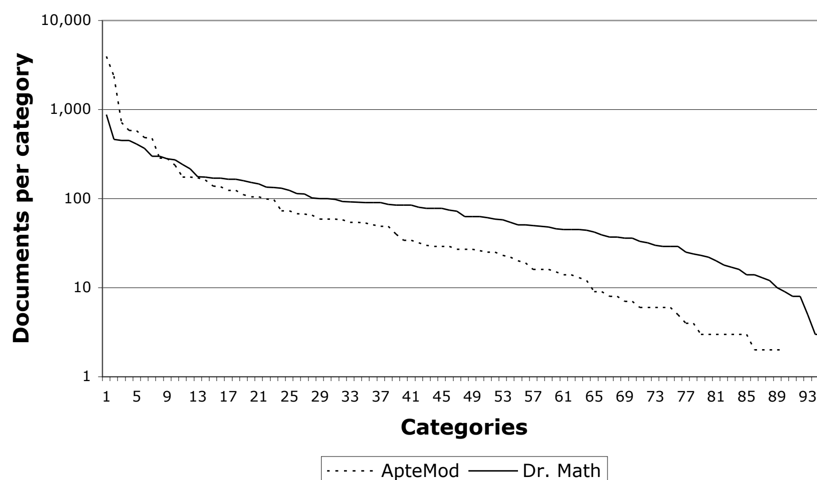

Figure 5.1: Category distributions for the test corpora

data source is even more skewed—in creating the corpus, any categories that did not contain at least one document in the training set and one document in the test set were removed from the corpus [50].

In the ApteMod corpus, each document belongs to one or more categories. There are 90 categories in the corpus. The average number of categories per document is 1.235, and the average number of documents per category is about 148, or 1.37% of the corpus.

Figure 5.1 shows the category distribution for the corpora discussed in this section. The categories are plotted on the horizontal axis, and the number of documents per category are plotted on the vertical axis using a logarithmic scale.

5.1.2 Dr. Math

The "Ask Dr. Math" service [16] is one of many so-called ask-an-expert services becoming common on the Internet. These services are generally staffed by domain experts who answer questions about the domain sent by non-experts.

Ask-an-expert services may be chiefly educational, as the Ask Dr. Math service is, or they may be tied to businesses' Customer Relationship Management or support initiatives.

Since the domain experts in these services often have a single subdomain of expertise, it is helpful if they can view only the subset of the questions that relate to that subdomain. However, the non-experts may not know how to properly categorize their own question when submitting it. For example, a user of the Ask Dr. Math service may be a middle school student unfamiliar with the particular mathematics taxonomy used by the service. It may therefore be infeasible for the users to categorize questions during submission, and it is in this situation that automatic categorization may be helpful.

The Dr. Math corpus is a collection of 6,630 English-language messages sent to the "Ask Dr. Math" ask-an-expert service for students [16]. Each message has been manually assigned by a domain expert to one or more categories, with category names indicating both math topic and grade level, e.g. "High School Geometry." There are 95 categories in the category set. The ontology is generally not separable into two separate category sets for independent topic and level categorizations, in part because many topic and level combinations like "Elementary School Calculus" don't exist in the category scheme.

The 26 MB corpus is divided into a training set with 5,304 documents and a test set with 1,326 documents. As with the ApteMod corpus, the category distribution is skewed, with 13.2% of the documents in the most common category and only 0.0452% (3 documents) in the least common category. The average number of categories per document is 1.534, and the average number of documents per category is about 107, or 1.61% of the corpus.

The Dr. Math corpus is an important data set for proving the effectiveness of the `AI::Categorizer` framework, because it represents a potential real-world application in the educational domain, one of the key target application areas of the Web Engineering Group at the University of Sydney. Work with the Dr. Math corpus has been previously published in [45].

The Dr. Math corpus is not available for direct download, but interested parties may contact the author for details.

5.2 Quality of Categorization

In order to evaluate the quality of the results generated by the `AI::Categorizer` implementation code, categorization experiments were performed on each of the corpora described in Section 5.1 using the measures defined in Sections 2.4 and 2.4.1.

Several experimental parameters (e.g. category membership thresholds) can be set for each experiment; for the ApteMod corpus these were set to match the parameters used in [50], where known. For the other corpora they were optimized to provide the best performance on the test set—in this sense, the test set might be thought of more correctly as a validation set, because in a strict testing environment the performance on the test set should not influence training parameters. This methodology was adopted because it more closely matches the procedure that would be used when building an application, in which the true test set may consist of documents not yet possessed by the developer, i.e. the set of target documents in the application domain. This is the evaluation method used in several studies in the literature, such as [21].

Where a baseline score is given in the results, this refers to a simple probabilistic categorizer that assigns categories to each document, weighting the probability of assignment by the frequency of each category in the training set. For instance, if a certain category was present in 40% of the training documents, any document in the test set would have a probability of 0.4 of being assigned to that category by the baseline categorizer.

It should be emphasized that this thesis does not claim to produce any new results in the area of developing Text Categorization algorithms. Descriptions of existing algorithms from the TC literature have formed the basis for developing the `AI::Categorizer` framework. The results presented here should be consid-

method	ρ^M	π^M	F_1^M	ρ^μ	π^μ	F_1^μ	error
NB	.3659	.4969	.3959	.7238	.8514	.7824	.00555
SVM	.0868	.3727	.1239	.5254	.9106	.6663	.00725
kNN	.2655	.3856	.2903	.7650	.7975	.7809	.00591
Baseline	.0135	.0142	.0137	.1645	.1664	.1654	.02287

Table 5.1: Results of `AI::Categorizer` on ApteMod corpus

method	ρ^M	π^M	F_1^M	ρ^μ	π^μ	F_1^μ	error
NB	-	-	.3886	.7688	.8245	.7956	.00544
SVM	-	-	.5251	.8120	.9137	.8599	.00365
kNN	-	-	.5242	.8339	.8807	.8567	.00385

Table 5.2: Results from [50] on ApteMod corpus

ered successful if they align with results already published in the literature—superior results should not be expected.

5.2.1 ApteMod

Table 5.1 summarizes the results of three different machine learning methods in `AI::Categorizer` as compared with the baseline categorizer described above. Table 5.2 gives similar scores from a well-known comparative study of common categorization algorithms [50]. Where possible, the present study has attempted to duplicate the findings in [50], though in some cases there is not enough information to duplicate the findings exactly, and in some cases the `AI::Categorizer` framework lacks certain features mentioned in [50]. These differences will be discussed below.

In the case of the Naïve Bayes categorizer (NB), the results match [50] closely, with a slightly greater F_1^M and a slightly smaller F_1^μ. To match the experimental settings in [50], the size of the feature-set was set to 2000.

For the Support Vector Machine categorizer (SVM), the performance is significantly worse than the findings in [50]. The reasons for this are not clear—`AI::Categorizer` currently uses an SVM implementation based on the `libsvm` C library [7], whereas [50] used SVM^{light} as its implementation [22], and there may be major differences in the behavior of these two libraries. Since the macro-

averaged scores are particularly bad, it may be inferred that [7] is not performing well on rare categories. In both studies, a linear SVM kernel was used, and 10,000 features were considered when building the categorization model.

One possible reason for the poor performance of the SVM categorizer is that for both micro- and macro-averaged scores, π is much larger than ρ, indicating that this situation could be balanced by choosing more appropriate category membership thresholds. However, `libsvm` doesn't seem to allow tuning of these thresholds, so a remedy for this situation isn't clear.

For the k-Nearest-Neighbor categorizer (kNN), F_1^μ results are comparable with NB, but no scores are as good as the kNN results in [50]. The major difference between the two implementations is that the implementation in [50] finds per-category membership thresholds by optimizing performance on a validation set, while the implementation in `AI::Categorizer` uses a single membership threshold for all categories, settable by the user. In this experiment the threshold was set to 0.1, a value which is not meaningful in itself, but which seemed to give the best performance. Note, that the macro-averaged precision is higher than recall, but the micro-averaged recall is higher than precision, indicating that it is not possible to simultaneously find the optimal threshold for both rare and common categories. Thus using individual thresholds for each category should increase F_1 scores if optimized properly. The k parameter in this experiment (indicating the number of similar documents to consider when categorizing) was set to 45, and the number of features considered was set to 2415, both to match the values used in [50].

The kNN implementation in `AI::Categorizer` is fairly new, and is an adaptation of work by another researcher. The ability to optimize per-category thresholds is considered a useful future addition to the framework, and will be added soon.

For all three categorizers, the `tfidf_weighting` parameter was set to `xfx`, indicating that words are weighted by the logarithm of their inverse document frequency. This is the same setting used in [50]. One other possible difference

method	ρ^M	π^M	F_1^M	ρ^μ	π^μ	F_1^μ	error
NB	.2338	.2677	.2358	.3766	.3542	.3651	.0215
SVM	.3333	.1562	.1946	.4211	.2273	.2952	.0330
kNN	.2372	.2824	.2154	.3607	.3572	.3590	.0212
Baseline	.0156	.0176	.0161	.0406	.0421	.0413	.0310

Table 5.3: Results of `AI::Categorizer` on Dr. Math corpus

between [50] and the current experiment is that [50] used either a χ^2 or *information gain* criterion (it is not clear which criterion was used with which categorizer) for feature selection, while *document frequency* was used here. This should not be a major factor in the results, as document frequency has been shown to produce results competitive with other feature selection methods on this corpus [51].

5.2.2 Dr. Math

Table 5.3 shows the categorization performance of `AI::Categorizer` on the Dr. Math corpus. Note that the baseline micro-averaged scores are much lower than on the ApteMod corpus (Table 5.1), indicating that this may be a more difficult categorization task simply because the category distribution is flatter than in ApteMod (Figure 5.1).

Because no other TC study has been done on the Dr. Math corpus, not much can be said about the comparative results of the `AI::Categorizer` framework. However, the noteworthy points will be discussed below.

Using the Naïve Bayes categorizer, the F_1 scores are about 0.24 when macro-averaged and 0.37 when micro-averaged. This is not as dramatic a difference as seen on the ApteMod corpus, where the micro-averaged scores are approximately double the macro-averaged scores. In this experiment, the number of features considered was set to 20% of the training corpus, or 1764 features, though varying this parameter between 1500 and 3000 seemed to produce similar results.

Using the SVM categorizer, F_1 scores were again worse than with NB, but in this case the recall with SVM was higher than the precision—the opposite

of the situation on the ApteMod corpus. Again, a mechanism for trading off ρ and π would be desirable. The best scores with SVM were obtained when using no feature selection at all, i.e. using all 8824 features when building the SVM models. This is an indication of SVM's robustness to noise in the data sets it considers.

With the kNN categorizer, results were competitive with the NB categorizer. Note that F_1^M for kNN is lower than each of ρ^M and π^M—this somewhat counterintuitive situation, unique to macro-averaging, can arise when the per-category scores ρ_i are sometimes greater than π_i and sometimes less than π_i. The best results for this experiment were found when using a feature set with 3000 features and $k = 15$.

5.3 Efficiency

In order to assess the efficiency of the implementations in `AI::Categorizer`, the running times and memory usage of the framework on the tasks in Section 5.2 are reported here. All tests were performed on a Sun Ultra-Enterprise server running Solaris 7, with a processor speed of 400 MHz, and with 2040 Megabytes of RAM. The software used was a pre-release version of `AI::Categorizer` version 0.05, running under perl version 5.6.1.

The reader is cautioned that the resource usage numbers presented here will be highly volatile from one system to another, and that they may even change significantly with different versions of the software running on the same system. The results presented here may give a rough indication of how the efficiency of different techniques relate to each other, however, and how the sizes of the data sets affect efficiency.

Each experiment is divided into four stages, each run as a separate system process:

Scan The training corpus $\mathcal{T}^r$ is scanned to determine the most relevant features, using the document frequency feature selection criterion, and the

		Scan	Read	Learn	Test
ApteMod	NB	128.6s	122.8s	22.9s	244.2s
	SVM	133.1s	126.6s	2332.3s	448.2s
	kNN	126.9s	121.8s	21.5s	2616.2s
Dr. Math	NB	44.8s	43.2s	7.2s	46.7s
	SVM	43.7s	42.8s	734.5s	118.9s
	kNN	43.8s	42.2s	6.6s	725.2s

Table 5.4: Time required for the experiments in Section 5.2. Measurements are given in CPU seconds.

		Scan	Read	Learn	Test
ApteMod	NB	10.5M	48.7M	47.0M	13.6M
	SVM	12.2M	50.6M	876.6M	31.1M
	kNN	10.8M	48.0M	64.4M	44.1M
Dr. Math	NB	8.7M	24.1M	24.1M	11.0M
	SVM	9.4M	26.3M	304.7M	21.5M
	kNN	9.0M	25.1M	28.1M	21.9M

Table 5.5: Memory usage for the experiments in Section 5.2. Measurements are given in megabytes of virtual memory usage.

list of relevant features is saved to disk.

Read The entire training corpus $\mathcal{T}^r$ is read into memory as a `KnowledgeSet`, using only the features determined in the "Scan" step, and saved to disk as a data structure accessible for the next step.

Learn A `Learner` object is created, and its `train()` method is invoked using the `KnowledgeSet` from the previous step. The `Learner` is saved for use in the next step.

Test The `Learner` from the previous step is loaded into memory, and each document from the test corpus $\mathcal{T}^e$ is categorized using the `Learner`'s `categorize()` method.

Table 5.4 shows the running times for each phase of the experiments conducted in Section 5.2, and Table 5.5 shows their memory usage. Running time is given in total CPU seconds consumed, and memory usage is measured by reporting the total size of the process in virtual memory.[1] Note that the run-

[1]On Unix operating systems, including the Solaris system on which these tests were per-

ning times include both the loading of any data structures from the previous stage and the saving of any data structures for the next stage. The memory usage includes any data structures used by the application as well as the perl interpreter itself.

Of course, the running times and memory usage during the "Scan" and "Read" stages do not vary much when the Machine Learning algorithm is changed, because this part of the process does not involve the `Learner` class. The numbers are presented so that the reader may compare the measurements on the different corpora, and any unexpected differences are likely due to the state of other concurrent processes on the machine, caching of disk data, and the like.

Among the most obvious characteristics of the results in Tables 5.4 and 5.5 is that the Naïve Bayes categorizer is very efficient in terms of both running time and memory. The SVM categorizer is slower during the "Learn" phase by a factor of about 100, and the kNN categorizer is slower during the "Test" phase by a factor of 10–20. This is consistent with the theoretical properties of the algorithms discussed in Section 2.3.

One other interesting property is that the SVM categorizer seems to take roughly twice as much time and memory as the NB categorizer during the "Test" phase. This is not necessarily supported by the theoretical properties of the two algorithms, so this finding may be due to implementation issues in the SVM code or the Adapter to the SVM engine.

5.4 Applications

Unfortunately, there are as yet no standard methods of objectively evaluating the quality of a framework's design. Such a method is not likely to appear in the near future, because framework design is a highly subjective process requiring

formed, the term "virtual memory" when describing the size of a process means the total size of the running process, including any memory segments in Random Access Memory and any cached to the disk. It does not refer just to the portion that may be resident on the disk cache, as the term is sometimes used to mean on Windows or older Macintosh systems.

much expertise in the framework developer [15, sec. 1.5]. Some quality metrics have been proposed, but they tend to rely on comparing different releases of a framework as it evolves to meet application needs, interpreting any major changes as flaws in the original framework design [15, ch. 25]. Because `AI::Categorizer` is a newly released framework, this kind of comparison between releases is not possible.

In order to provide evidence of the framework's usefulness, therefore, one option is simply to build different kinds of applications using the framework and judge whether the applications were successful or not, and whether the framework seemed well-suited for the application. Ideally, the applications will represent at least some of the use cases that the framework was designed to support. This is what will be done here. Three applications using `AI::Categorizer` that were developed during the course of the candidature on which this thesis is based will be discussed in the following sections.

5.4.1 Command-line categorizer

The first framework is a simple command-line script, distributed with the framework code, which provides a relatively simple way to instantiate a top-level `AI::Categorizer` object and invoke several of its methods. Its main purpose is to support the kinds of scientific investigations described in Section 3.2.1.

Using this script, the user may specify the parameters for an experiment either directly on the command line, or in a file of key-value pairs indicating the parameter's name and value. A sample file of this sort is shown in Figure 5.2.

This command-line application also provides support for storing the sets of parameters and the categorization results to disk. In this way, the user keeps a historical record of which parameter settings improved performance and which had negative effects.

This application contains only 110 lines of Perl code, most of which deals with benchmarking issues or parsing of the command-line options. This shows that a data-driven application can be built on top of the framework fairly easily.

```
--- #YAML:1.0
data_root: corpora/drmath-1.00
stopword_file: corpora/drmath-1.00/SMART.stoplist
progress_file: "/tmp/drmath-NB"
outfile: "results/drmath-NB.txt"
learner_class: "::NaiveBayes"
tfidf_weighting: xfx
stemming: porter
features_kept: 0.2
verbose: 1
```

Figure 5.2: Parameter specification file for testing the Naïve Bayes categorizer on the Dr. Math corpus.

5.4.2 Database categorization

In order to support the categorization of documents inside a database as mentioned in Sections 3.2.2 and 3.2.4, David Bell of the Web Engineering Group at the University of Sydney developed an application which embeds `AI::Categorizer` directly in the PostgreSQL database engine. Bell's work involved the kNN categorizer, but because the framework was used instead of a custom kNN implementation, any of the framework's categorizers could be used in the same manner.

Using this application, categorization functionality was made available in database insertions and queries through the use of embedded functions like `categorize()`. This allows categorization of arbitrary text in the database, because the input to the categorization function is any string that can be built using the database query language.

This application demonstrates the ability of the `AI::Categorizer` framework to provide functionality in embedded environments.

5.4.3 Client-server categorization

As part of a research project in the financial domain, Mark Aufflick of the Web Engineering Group at the University of Sydney created a categorization

server like that described in Section 3.2.3. In this application, the categorizer resides in a long-lived server daemon process that accepts XML/RPC requests for categorizing arbitrary documents. The server responds with the categorization information as an XML/RPC response, which may contain the categories assigned or the single best category assigned.

The application was built using standard Perl tools for constructing XML/RPC servers, available from the CPAN. This demonstrates the ability of the `AI::Categorizer` framework to build custom categorization servers for diverse application needs.

Chapter 6

Conclusion

This thesis has presented the background, design, implementation, and evaluation of a new object-oriented application framework for Text Categorization. The framework has been designed to facilitate novel work in Text Categorization research and rapid development of TC applications. The framework does not include any novel work in Text Categorization algorithms, but it has been designed with the intention of facilitating such novel work.

6.1 Evaluation and Outcomes

The evaluation in Chapter 5 shows that the Naïve Bayes implementation in `AI::Categorizer` gives results consistent with the work of others in the TC literature, and that the k-Nearest-Neighbor and Support Vector Machine implementations are not yet comparable with the state-of-the-art implementations of the algorithms referenced in the literature. It also shows the relative efficiencies of the three algorithms during the four main execution phases of a TC application: the Naïve Bayes learner is fairly efficient in all phases, the SVM learner requires significant resources during the training phase, and the kNN learner requires significant resources during the test categorization phase. This confirms facts about these core algorithms known from their research literature.

The architectural quality of the framework is difficult to evaluate directly, but several analysis approaches have been presented in this thesis. Several aspects of the framework's design have been presented as instances of Design Patterns, indicating that they may align with current best-practices in framework development. Example applications of several types have also been discussed, indicating that the framework can support the use cases presented in Section 3.2.

6.2 Further Work

Since one of the main goals of framework development is that the framework should be easily extensible and developers can add new functionality with a minimum of effort, there are many avenues for further work with the framework. For instance, adding common functionality in the core framework is helpful as the framework progresses from a whitebox to a more blackbox style of usage.

A selection of some of the most desirable directions for further work is presented here, in no particular order.

- As mentioned in Section 4.3.1, different feature vector data structures may be desirable in different situations. To provide framework users with the flexibility to make trade-offs between speed, memory, and complexity, the sparse `C`-level structure described there should be implemented.
- Many new algorithms are continually being developed and refined in the TC literature, and the most promising and general-purpose of these should be incorporated into the framework.
- In order to allow application developers to use and extend the framework more easily, different kinds of documentation, such as tutorials and recipes, should be written [15]. suggests that multiple kinds of documentation can aid framework adoption.
- The current Support Vector Machine and k-Nearest-Neighbor categorizers

should be investigated and improved so that they deliver performance matching the published results in the well-known TC literature.

- It might be helpful if the functionality in the Naïve Bayes categorizer were available as a separate module outside the framework, so that developers interested in Machine Learning problems outside the TC domain could apply the algorithms it implements. The implementation in `AI::Categorizer` could then become a simple Adapter to the outside module.

- The framework should be put to use in a hierarchical categorization application, with possible hierarchical functionality added to the framework to support such usage.

- Alternate feature selection algorithms, such as the χ^2 and IG algorithms discussed in Section 2.2.5, should be implemented and added to the core framework.

Bibliography

[1] L. Douglas Baker and Andrew Kachites McCallum. Distributional clustering of words for text classification. *Proceedings of SIGIR-98, 21st ACM International Conference on Research and Development in Information Retrieval*, 1998.

[2] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Pub Co, 1998.

[3] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/, 2000.

[4] Chris Buckley, Gerard Salton, James Allan, and Amit Singhal. Automatic query expansion using SMART: TREC 3. In *Text REtrieval Conference*, pages 0–, 1994.

[5] Rafael A. Calvo and Ken Williams. Automatic categorization of announcements on the australian stock exchange. *7th Australasian Document Computing Symposium*, 2002.

[6] Soumen Chakrabarti, Byron Dom, Rakesh Agrawal, and Prabhakar Raghavan. Scalable feature selection, classification and signature generation for organizing large text databases into hierarchical topic taxonomies. *VLDB Journal: Very Large Data Bases*, 7(3):163–178, 1998.

[7] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: a library for support vector machines (version 2.36).

[8] Damian Conway. *Object Oriented Perl*. Manning Publications Company, August 1999.

[9] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[10] Serge Demeyer, Theo Dirk Meijler, Oscar Nierstrasz, and Patrick Steyaert. Design guidelines for tailorable frameworks. *Communications of the ACM*, 40(10):60–64, 1997.

[11] Chris Dibona, Mark Stone, and Sam Ockman, editors. *Open Sources: Voices from the Open Source Revolution*. O'Reilly and Associates, 1 edition, January 1999.

[12] Pedro Domingos and Michael Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29(2-3):103–130, 1997.

[13] Susan T. Dumais and Hao Chen. Hierarchical classification of web content. *Proceedings of SIGIR 2000*, 2000.

[14] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[15] Mohamed Fayad and Douglas C. Schmidt, editors. *Building Application Frameworks*. John Wiley & Sons, 1999.

[16] The Math Forum. Ask Dr. Math web site. http://www.mathforum.org/dr.math/.

[17] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly and Associates, 2 edition, July 2002.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[19] Åsa Granlund, Daniel Lafière, and David A. Carr. A pattern-supported approach to the user interface design process. *HCI 9th International Conference on Human-Computer Interaction*, August 1999.

[20] Warren R. Greiff and Jay M. Ponte. The maximum entropy approach and probabilistic IR models. *ACM Trans. on Information Systems*, 18(3):246–287, 2000.

[21] Thorsten Joachims. Text categorization with support vector machines: learning with many relevant features. In Claire Nédellec and Céline Rouveirol, editors, *Proceedings of ECML-98, 10th European Conference on Machine Learning*, number 1398, pages 137–142, Chemnitz, DE, 1998. Springer Verlag, Heidelberg, DE.

[22] Thorsten Joachims. *Making Large-Scale SVM Learning Practical*, chapter 11. MIT Press, 1999.

[23] Ralph E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, 1997.

[24] Daphne Koller and Mehran Sahami. Hierarchically classifying documents using very few words. In Douglas H. Fisher, editor, *Proceedings of ICML-97, 14th International Conference on Machine Learning*, pages 170–178, Nashville, US, 1997. Morgan Kaufmann Publishers, San Francisco, US.

[25] David D. Lewis. Naive (Bayes) at forty: The independence assumption in information retrieval. In Claire Nédellec and Céline Rouveirol, editors, *Proceedings of ECML-98, 10th European Conference on Machine Learning*, number 1398, pages 4–15, Chemnitz, DE, 1998. Springer Verlag, Heidelberg, DE.

[26] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.

[27] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[28] Dunja Mladenic. Feature subset selection in text-learning. In *European Conference on Machine Learning*, pages 95–100, 1998.

[29] Ted Pedersen. Machine learning with lexical features: The Duluth approach to Senseval-2. *Proceedings of SENSEVAL-2: Second International Workshop on Evaluating Word Sense Disambiguation Systems*, July 2001.

[30] John C. Platt. *Fast training of support vector machines using sequential minimal optimization*, chapter 12, pages 185–208. MIT Press, 1999.

[31] J. R. Quinlan and R. L. Rivest. Inferring decision trees using the minimum description length principle. *Information and Computation*, 80(3):227–248, 1989.

[32] Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly and Associates, 1997.

[33] Dave Rolsky and Ken Williams. *Embedding Perl in HTML with Mason*. O'Reilly and Associates, 2002.

[34] Ivan A. Sag and Thomas Wasow. *Syntactic Theory: A Formal Introduction*. Cambridge University Press, 1999.

[35] Gerard Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley, Reading, Pennsylvania, 1989.

[36] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management: an International Journal*, 24(5):513–523, 1988.

[37] Bernhard Schölkopf and Alex Smola. *Learning with Kernels*. MIT Press, 2002.

[38] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys (CSUR)*, 34(1):1–47, 2002.

[39] Kagan Tumer and Joydeep Ghosh. Error correlation and error reduction in ensemble classifiers. *Connection Science*, 8(3):385–404, December 1996.

[40] UserLand Software, Inc. XML-RPC Home Page. http://www.xmlrpc.com/, 2003.

[41] C. J. Van Rijsbergen. *Information Retrieval, 2nd edition*. Butterworths, 2 edition, 1979.

[42] Various authors. Comprehensive Perl Archive Network. http://www.cpan.org/, 2003.

[43] C. S. Wallace and John D. Patrick. Coding decision trees. *Machine Learning*, pages 7–22, 1993.

[44] Ken Williams and Rafael A. Calvo. A framework for text categorization. *7th Australasian Document Computing Symposium*, 2002.

[45] Ken Williams, Rafael A. Calvo, and David Bell. Automatic categorization of questions for a mathematics education service. In *Proceedings of the 11th International Conference on Artificial Intelligence in Education*, August 2003.

[46] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, 1990.

[47] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, chapter 8, pages 265–320. Morgan Kaufmann Publishers, 1 edition, 1999.

[48] World Wide Web Consortium DOM Working Group. Document Object Model (DOM) Level 1 Specification. http://www.w3.org/TR/REC-DOM-Level-1/, 1998.

[49] Yiming Yang. A study on thresholding strategies for text categorization. In *SIGIR*, 2001.

[50] Yiming Yang and X. Liu. A re-examination of text categorization methods. In *22nd Annual International SIGIR*, pages 42–49, Berkley, August 1999.

[51] Yiming Yang and Jan O. Pedersen. A comparative study on feature selection in text categorization. In Douglas H. Fisher, editor, *Proceedings of ICML-97, 14th International Conference on Machine Learning*, pages 412–420, Nashville, US, 1997. Morgan Kaufmann Publishers, San Francisco, US.

Appendix A:
A Framework for Text Categorization

Ken Williams
Web Engineering Group
The University of Sydney
Bldg J03, Sydney NSW 2006
kenw@ee.usyd.edu.au

Rafael A. Calvo
Web Engineering Group
The University of Sydney
Bldg J03, Sydney NSW 2006
rafa@ee.usyd.edu.au

Abstract

In this paper we discuss the architecture of an object-oriented application framework (OOAF) for text categorization. We describe the system requirements and the software engineering strategies that form the basis of the design and implementation of the framework. We show how designing a highly reusable OOAF architecture facilitates the development of new applications. We also highlight the key text categorization features of the framework, as well as practical considerations for application developers.

Keywords Document Management, Text Categorization, Application Frameworks

1 Introduction

Automatic Text Categorization (TC) has been an active research area for over a decade and is increasingly being used in the development of commercial applications. These commercial applications usually belong to one of two system types: in-house systems implemented in order to solve a particular company's specific problems, and generic systems marketed to corporations as ready-made categorization solutions. The former tend to be ad-hoc solutions not suitable for use by others, and not made publicly available. The latter tend to be proprietary, closed-source, expensive solutions inaccessible to individuals, small companies, and researchers.

One result of this situation is that many techniques and design strategies are underdeveloped as they are not well-known to research or application communities. Systems such as Weka [14] or Libbow [7] are widely used by the research community, but tend not to focus on integration into real-world applications. By contrast, the commercial systems are often useless for research because they are closed-source, generalize poorly to new problems, or cost more than most researchers can afford. Therefore, researchers do not get the benefit of leveraging industry's TC applications, and industry doesn't get the benefit of the latest developments and knowledge from the research community.

Proceedings of the 7th Australasian Document Computing Symposium,
Sydney, Australia, December 16, 2002.

It is our aim to create first-rate customizable tools for Text Categorization that apply equally well to the problems of industry and research. Our tools should also be accessible to the casual or small-time developer interested in TC. To accomplish this, we have implemented a framework for Text Categorization.

Before discussing the details of the framework, we will briefly look at some general background on frameworks. Different software engineering architectures are used for different sets of requirements. The most common kinds of software architectures include:

Applications Application developers focus on improving internal reusability and interfacing with users. Developer or user extensibility need not be considered–the application is considered complete as delivered. A popular example of a classification application is the Weka Machine Learning system [14].

Toolkits and libraries Library developers focus on generic reusability for multiple applications. Examples include the mathematical or networking libraries that exist for most programming languages. The "bow" library [7] is an example from TC. Developers who use a library do not have to learn its internal architecture, and the library does not dictate the structure of the application under development.[4] The internal implementation of the library is considered to be hidden from its users.

Frameworks A framework is a set of classes that embodies an abstract design for solutions to a family of related problems [4, Ch. 2]. Framework designers focus on applicability to a certain set of problems, and on flexible best-practices embodied in software. An "inversion of control" puts the framework in charge at a high level inside the application, with custom application code playing a

subordinate role–therefore, interfaces between framework classes must be documented and stable. Common examples of frameworks include generic application frameworks like Apple's "Cocoa." Weka may also be considered a framework when it is used to implement new categorization algorithms through subclassing.

Before deciding on one of these approaches it is important to define the main user audience for text categorization systems in order to determine requirements for a useful TC system. We see typical TC users in terms of the following roles:

Application Developer A professional such as a web developer or engineer that needs to add automatic categorization features to a software application. The application developer may have no prior experience with Text Categorization. The end user may have varying degrees of control over the categorization process.

Researcher A TC researcher interested in novel approaches to machine learning or document processing. This professional is often not interested in implementing a real world application, but wishes to improve existing TC algorithms and methodologies.

Domain Expert Complex applications often require a domain expert who dictates project requirements and has expertise in the application domain (e.g. financial documents, knowledge management). The domain expert often makes high-level decisions about when TC could be effective in the given domain, and needs to exert fine control over the TC process.

Of course, one person may play several of these roles simultaneously.

A researcher will most often want to use a TC system as a framework, because they need to integrate custom code into the system at a low level. A researcher may also find it convenient to use a TC system as an application which provides a convenient user interface for running common kinds of experiments. By contrast, an application developer may want to use a TC system as a library or set of libraries, providing no custom code of his or her own.

Given these requirements, we decided to implement our software as a framework rather than as an application or set of libraries. One reason for this is that a framework can easily be turned into an application by providing simple wrapper code, and it can be turned into a library by providing concrete implementation classes. However, libraries and applications can not typically be turned into frameworks very easily. Therefore, a framework provides the best coverage for the perceived needs of the TC community.

The framework described in this paper includes classes for managing documents, collections of documents, categorization algorithms, and so on. The core framework includes both concrete classes like "Naïve Bayes Learner" which may be used without custom development, as well as abstract classes like "Boolean Learner" which require the user to implement certain behaviours before using them. Abstract classes provide a starting point and an interface for new development and reduce repeated work.

2 Design Requirements

A framework must be able to accommodate functionality in a number of essential areas, providing common behaviour while allowing users and developers to customize behaviour through configuration parameters and/or framework subclassing. We summarize the design issues in this framework as follows. Note that some of these issues are general framework design issues, while others are more specific to this particular domain.

Framework reusability The main reason for building a framework rather than a single text categorization application is to increase reusability of design and implementation. Framework research literature provides guidelines on building application frameworks.[4]

Modularity The components' internal implementations should be able to change without affecting the other components.

Integration The framework should be able to interface easily with existing categorization solutions (e.g. Weka, libbow, various Neural Net libraries, and feature selection packages), uniting many solutions under a common interface.

Rapid Application Development Prototyping new applications should be very quick, with a minimum of custom code in each case. Custom code should generally implement new behaviors rather than new structures within the framework.

Rapid Research Cycle Researchers should be able to quickly investigate new questions, using the framework as a starting point.

Model Flexibility The framework structure should be flexible enough to accommodate the needs of many different categorization algorithms that may operate on different representations of the underlying data.

Computational Efficiency The data sets involved can be quite large, so it is important to have a design and implementation that is efficient in memory, CPU time, and other practical measures such as the time it takes to load a categorizer from disk and generate a hypothesis.

Separability Pieces of the framework should be usable in isolation for users that only need a feature selection package, a vector categorizer, etc. The most separable pieces of the framework should in many cases be completely separated and available under separate distribution, and used as a software dependency in our framework.

With the above issues in mind, we have chosen to implement the framework using the Perl programming language. [13] A vast number of Perl modules are freely available for many different tasks, which extends the domain of applicability for the framework. Many of these modules are tools for processing text, and can be used by the framework. Perl is widely used, multi-platform and integrates well with other languages, so it enables fast prototyping. Perl is also natively object-oriented, with a very flexible object model.[2]

In Perl, the basic unit of reusable code is called a *module*; our framework is implemented as the `AI::Categorizer` module.

3 Functional Areas

The framework supports several functional areas of Text Categorization. We describe them here together with the tradeoffs and design decisions that may be useful to other researchers developing TC systems.

Figure 1 shows the architecture of the framework. Attributes and methods of each class have been removed for the sake of brevity. Each of the classes will be discussed in the context of their Text Categorization function. `Categorizer` is the top level class, which manages the data-related classes (`KnowledgeSet`, `Collection`, `Document` and `Category`), as well as the machine learning `Learner` classes and `Hypothesis`, and a class for reporting the results.

Data format

Since documents come in a wide variety of formats such as XML, plain text, or PDF, the framework should support the importing of knowledge in several formats and have a mechanism by which the user may extend these capabilities for a particular environment. The base class `Document` allows the user to specify the content as a string. The user may also subclass the `Document` class, overriding the `parse()` method for direct importing of data in its natively stored format.

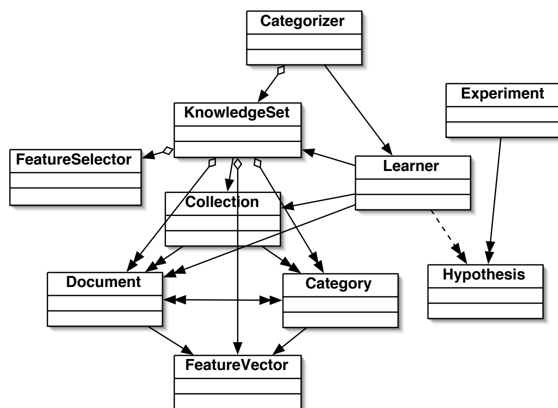

Figure 1: Simplified UML class diagram for the framework

In the `Collection` class and its subclasses, the framework also supports the notion of a collection of stored documents, such as a directory of text files, a database of stored documents, or an XML file containing multiple documents. The most common storage formats can be a part of the core framework, while proprietary or unusual formats can be implemented through subclassing. Note that the document format and collection format are independent characteristics; a project may have a directory of text files, a directory of XML files, or a directory of PDF files, but these would all be handled by the `Collection::Files` class with the appropriate `Document` subclass. Likewise, a project may have a collection of XML documents stored in a single file, as a directory of files, or in a database, but these would all be parsed by the `Document::XML` class with the appropriate `Collection` subclass. Note that `Document` and its subclasses exist mainly for the purpose of importing data; after the data is read and parsed, the rest of the system will throw away most of the information in the `Document` object, keeping only its `FeatureVector` object and the list of `Categories` associated with the `Document`.

Structured documents

Each document may have several sections of content, such as “body”, “subject”, “signature”, and so on. In `AI::Categorizer`, the user specifies the content by providing a hash of key-value pairs, where the key indicates the name of the section, and the value is a string containing the content data. The user may also specify “weights” to assign to the features found in each section. In the future, other treatments for the different sections of a document may be supported as we develop effective ways to use this structure.

Tokenizing of data

The default implementation tokenizes document data by extracting all non-whitespace byte sequences between word-character boundaries. This is usually sufficient in English, but non-English language documents or documents with unusual content will certainly necessitate custom tokenization. To achieve this, the user may subclass the `Document` class and override its `tokenize()` method if a different algorithm is required. We may also add other tokenizing options to the default implementation, controlled by parameters, if other common tokenizing needs are found.

Linguistic stemming

The default implementation provides support for the Porter stemming algorithm, a standard algorithm for removing morphemes from English words to obtain their "stems," or root forms. By default no stemming is performed, but a `stemming` parameter can be set to `porter` to activate stemming. Alternatively, the user may override the `stem_words()` method of the `Document` class for custom stemming. This may be extremely important in highly morphological languages or in certain application domains.

Feature selection

Feature selection is handled by the abstract `FeatureSelector` class and its concrete subclasses. These classes implement `scan_features()` and `select_features()` methods. The `select_features()` method works on an entire `KnowledgeSet` in-memory at once. The `scan_features()` method can scan a collection of documents for the best features without necessarily loading the entire collection into memory. Both methods return a `FeatureVector` object to the client (typically a `KnowledgeSet`), which saves the list of highest-ranking features to use when parsing future documents.

The default implementation uses a simple Document-Frequency criterion for selecting features to use in model-building and categorization. This is very efficient, and has been shown in [16] to be competitive with more elaborate criteria in many common situations. We will add more criteria as the project develops.

Vector space modeling

The full range of TF/IDF weighting from [11] are supported, controlled by a `tfidf_weighting` parameter. If the user wants to employ a different weighting scheme, the `weigh_features()` method in the `KnowledgeSet` class may be overridden.

Machine Learning algorithm

Choosing a machine learning algorithm is done by choosing a subclass of the `Learner` class. Several algorithms have already been implemented including Naïve Bayes [6], Support Vector Machines [12] [3], Neural Networks [1] [15], k-Nearest Neighbors [15], and Decision Trees [9]. Any `Learner` class needs to implement the virtual methods `create_model()` and `get_scores()`, which supply the semantics behind the `train()` and `categorize()` methods, respectively. Since many Machine Learning algorithms are implemented as a series of binary decisions concerning individual category memberships, an abstract `Learner::Boolean` class is provided to help developers of new categorizers–in this case, one need only implement the smaller `create_boolean_model()` and `get_boolean_score()` methods.

Note that the `Learner` class does dual duty as a learner and a categorizer. No class distinction is made in the framework between a `Learner` before and after it has been trained–they are objects of the same class. This allows for the possibility of on-line learning, in which a trained learner incrementally uses additional training examples to improve its current model.

Machine Learning parameters

Because each ML algorithm may have several implementation parameters to control behavior, each `Learner` subclass accepts different parameters. To facilitate the wide variety of parameters that different classes may require, we use the `Class::Container` module[1]. This module allows each `Learner` subclass to declare the parameters it accepts, so that a Neural Network class can declare arguments for number of input, hidden, and output nodes, a k-Nearest Neighbor class can declare arguments for k and for thresholding strategies, and so on. These parameters are passed through the framework transparently using a variation on the "Factory Method" pattern. [5]

In fact, the `Learner` and its subclasses are not the only pieces of the framework in which varying parameters control operations. Because this situation is common throughout the framework, `Class::Container` is employed consistently for all structural classes in the framework. This goes a long way toward reducing the number of classes necessary to implement varying behavior.

Hypothesis behavior

Certain applications (e.g. newswire categorizers) may need to find "all categories that apply" for each document, whereas other applications (e.g.

[1] available at http://search.cpan.org/author/KWILLIAMS/Class-Container-0.08/

automatic email routers) may only be interested in the "best N categories," where N is often 1. These scenarios are supported by the Hypothesis class, which provides a generic interface to the scoring decisions of the categorizers. Methods like `categories()`, `best_category()`, and `in_category()` provide application-level access to categorization decisions based on the scores assigned by the `Learner` class.

On-line training

Some machine learning algorithms can easily integrate new knowledge into the knowledge base without going through the potentially expensive process of re-training the categorizer from scratch. For instance, most kNN implementations can do this, whereas most Neural Network implementations cannot. For categorizers that support this, a virtual `add_knowledge()` method in the `Learner` class is supplied. Currently no `Learner` subclasses in `AI::Categorizer` support on-line learning, but the architecture supports it when an implementation is needed.

4 Framework Customization

Like C++ and Java, Perl is natively object-oriented, but unlike them it does not have strict separation of compilation and execution stages. Rather, the compiler and interpreter work in tandem, trading back and forth to execute a Perl application, allowing runtime compilation of code. In addition, Perl's object model is fairly loosely bound (similar in this respect to Objective-C's model), permitting class names to be stored in variables and/or specified at runtime. Because of these properties, the choice of specific classes to be used in the framework can be made at runtime, controlled by parameters, facilitated by the `Class::Container` module. It allows several classes to cooperate as a framework without having to know about each others' class names, constructor parameters, and so on, and provides the glue to do strict early checking of parameter names and types, facilitating transparent factory patterns within the framework.

For instance, to use the built-in SVM learner, one could either create an `AI::Categorizer::SVM` object directly, or one could specify the class name by providing it as a value for the `learner_class` parameter. This behavior is implemented at the framework level, so different `Document`, `Collection`, `FeatureVector`, etc. classes can be pressed into service by the `document_class`, `collection_class`, and `feature_vector_class` parameters, respectively. This helps facilitate quick architectural changes, letting developers drop their own subclasses into the framework with relative ease.

5 Evaluation

Although the focus of this paper is the framework discussion and design, we present here some basic evaluation of its performance. We have evaluated our framework by building classifiers in several applications. We have implemented Naïve Bayes, Support Vector Machine, k-Nearest-Neighbor, and Decision Tree classifiers in the framework. We have trained classifiers using the standard Reuters ApteMod corpus and obtained similar results to the ones described in [15]. We have also trained and tested classifiers on other corpora in financial, educational, and discussion group domains. Due to space constraints and the proprietary nature of some of our other corpora, we will only describe results on the Reuters ApteMod corpus here, using the Naïve Bayes algorithm.

In training categorizers, we typically use two passes through the corpus when loading the data. The first pass scans the documents in order to perform feature selection, while the second pass actually loads the data into memory. This allows memory to be used more effectively than if we only made one pass over the data, because we avoid loading extraneous features. On the Reuters corpus, using Porter stemming [8] and a standard list of stopwords [10], the first pass over the 7769 training files may take roughly 59 CPU seconds and consume 11 MB of memory, while the second pass takes about 57 CPU seconds and consumes 32 MB. The memory figures reflect the total size of a running program, not just the size of the document data in memory.[2]

After the data is loaded, we pass it to a Learner object for training. Our Naïve Bayes training process takes 8.1 CPU seconds and consumes 40 MB of memory. Categorizing the 3019 test documents takes about 95 CPU seconds and consumes 14 MB.

With experimental settings similar to the ones described in [15] (we used Document Frequency feature selection, since we have not yet implemented χ^2 or Information Gain selection algorithms), we achieve recall, precision, and F_1 scores of 0.724, 0.851, and 0.782 when micro-averaged, and 0.366, 0.497, and 0.396 when macro-averaged. We believe any discrepancies with [15] are due to differences in feature selection and/or document tokenizing, but we have not tested this belief thoroughly.

6 Integration and Further Work

The framework has been used in a number of applications including an extension to the SQL language of the PostgreSQL relational database. It has also

[2]Tests were performed on a machine with a Pentium III 800Mhz chip, running Red Hat Linux release 7.0 and Perl 5.6.1. Results are not comparable across different architectures, but may be useful as a rough guide.

been used as distributed service for classification using an XML/RPC architecture, and integrated into multi-tier web applications and desktop applications.

We know that much work has been done by previous developers and researchers in the area of Text Categorization. While we are in one sense retreading ground by implementing generic TC software, we see our work as a way to extend the reach of others' work, rather than as a replacement for it.

To this end, we have tried to make the framework very inter-operable and provide interfaces to existing TC products. For instance, we have implemented a `Learner` subclass called `Learner::Weka` to provide an interface to any Weka classifier the user would like to use. In this way, `AI::Categorizer` benefits when progress is made in Weka, as well as the other way around.

We hope to create interfaces to other existing products as well. If the `AI::Categorizer` project gains enough momentum that other people wish to contribute code to it, we will encourage this code to be as independent and generic as possible so that we may simply create an interface to it in our framework. For instance, this is how the SVM learner in our framework was created recently–our `AI::Categorizer::SVM` class is just a thin wrapper around a generic `Algorithm::SVM` module by another author we collaborated with, and this in turn is a wrapper around the C library `libsvm`.

It is our hope that this strategy will extend the reach of both our framework and related existing and new TC software.

In designing the `AI::Categorizer` framework architecture, we have focused on aspects of Text Categorization that tend to remain common from one task to the next, allowing for growth in aspects that tend to change. For instance, we have specified that document features are encapsulated in a `FeatureVector` object, but we have not specified that object's internal implementation. Likewise, we have specified that the Machine Learning TC algorithms are encapsulated by the `Learner` class, but the specific algorithms will tend to vary from task to task.

In the first public versions of the framework, we have tended to implement the simplest versions of each of these classes, with more elaborate or optimized implementations deferred to later work. For instance, our `FeatureVector` class is currently implemented using Perl hashes, but other implementations (for instance, using C structs to implement sparse integer vectors) may be implemented in order to improve memory usage and/or speed. Other `Learner` subclasses may also be added, and the existing subclasses may be improved to provide more feature-rich implementations or improve efficiency. Because they are encapsulated in subclasses, these implementations may be traded at will, allowing experimentation with different implementations. In particular, we expect the `Learner` and `FeatureSelector` areas of the framework to grow as new algorithms are added and existing algorithms are informed by current research.

7 Conclusions

We have developed a new framework for Text Categorization which is publicly available and leverages existing work as much as possible. We have primary goals of providing usable TC software for application developers, researchers, and domain experts, as well as providing bridges between existing and new TC software. Our framework design endeavors to embody the key requirements which are common to most work in TC, and thus should improve reusability of design and implementation in applications that use text categorization. The analysis of its architecture may be useful to those embarked in building their own TC systems, so we have discussed the design decisions of the different functionalities supported by the framework.

Periodic point-releases of `AI::Categorizer` are available at http://www.cpan.org/modules/by-authors/id/KWILLIAMS/ , and bleeding-edge development versions are available via CVS at http://www.sourceforge.net/projects/ai-categorizer/ .

References

[1] Rafael A. Calvo. Classifying financial news with neural networks. In *6th Australasian Document Symposium*, page 6, December 2001.

[2] Damian Conway. *Object Oriented Perl*. Manning Publications Company, August 1999.

[3] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, Volume 20, Number 3, pages 273–297, 1995.

[4] Mohamed Fayad and Douglas C. Schmidt (editors). *Building Application Frameworks*. John Wiley & Sons, 1999.

[5] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[6] David D. Lewis. Naive (Bayes) at forty: The independence assumption in information retrieval. In Claire Nédellec and Céline Rouveirol (editors), *Proceedings of ECML-98, 10th European Conference on Machine Learning*, number 1398, pages 4–15, Chemnitz, DE, 1998. Springer Verlag, Heidelberg, DE.

[7] Andrew Kachites McCallum. Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering. http://www.cs.cmu.edu/ mccallum/bow, 1996.

[8] M.F. Porter. An algorithm for suffix stripping. *Program*, Volume 14, Number 3, pages 130–137, 1980.

[9] J. R. Quinlan and R. L. Rivest. Inferring decision trees using the minimum description length principle. *Information and Computation*, Volume 80, Number 3, pages 227–248, 1989.

[10] Gerard Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer.* Addison-Wesley, Reading, Pennsylvania, 1989.

[11] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management: an International Journal*, Volume 24, Number 5, pages 513–523, 1988.

[12] Bernhard Schölkopf, Christopher J.C. Burges and Alexander J. Smola (editors). *Advances in Kernel Methods – Support Vector Learning.* MIT Press, 1999.

[13] Larry Wall, Tom Christiansen and Jon Orwant. *Programming Perl.* O'Reilly and Associates, 3 edition, 2000.

[14] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Chapter 8, pages 265–320. Morgan Kaufmann Publishers, 1 edition, 1999.

[15] Y. Yang and X. Liu. A re-examination of text categorization methods. In *22nd Annual International SIGIR*, pages 42–49, Berkley, August 1999.

[16] Yiming Yang and Jan O. Pedersen. A comparative study on feature selection in text categorization. In Douglas H. Fisher (editor), *Proceedings of ICML-97, 14th International Conference on Machine Learning*, pages 412–420, Nashville, US, 1997. Morgan Kaufmann Publishers, San Francisco, US.

Appendix B: Automatic Categorization of Announcements on the Australian Stock Exchange

Rafael A. Calvo
Web Engineering Group
The University of Sydney
Bldg J03, Sydney NSW 2006
rafa@ee.usyd.edu.au

Ken Williams
Web Engineering Group
The University of Sydney
Bldg J03, Sydney NSW 2006
kenw@ee.usyd.edu.au

Abstract

This paper compares the performance of several machine learning algorithms for the automatic categorization of corporate announcements in the Australian Stock Exchange (ASX) Signal G data stream. The article also describes some of the applications that the categorization of corporate announcements may enable. We have performed tests on two categorization tasks: market sensitivity, which indicates whether an announcement will have an impact on the market, and report type, which classifies each announcement into one of the report categories defined by the ASX. We have tried Neural Networks, a Naïve Bayes classifier, and Support Vector Machines and achieved good results.

Keywords Document Management, Document Workflow

1 Introduction

The Australian Stock Exchange Limited (ASX - http://www.asx.com.au/) operates Australia's primary national stock exchange. Companies listed on ASX are required under the Listing Rules to make announcements about their activities "in order to ensure a fully informed market is maintained." [1] In order to guarantee access to this information, stock exchanges such as the ASX publish all recent and historical company announcements. Thanks to language technologies such as automatic document categorization, these corporate announcements can provide new sources of valuable financial information.

Historically, corporate announcements have provided valuable information to traders and the general public for decades. For this reason these announcements are used by regulators as the main tool to keep the market informed of all important events. The law assumes that these public announcements contain all the information needed by an individual trader to keep a reasonable understanding of what is happening with a particular company. This allows investors to make decisions based on information that is up to date and is equivalent to the information that company insiders might have. There is little doubt about the value of the information contained in these announcements, and several research groups are developing novel applications using this data. We describe in this article the evaluation of categorization techniques used to build these applications.

The ASX Data Services is a financial information service providing daily market information from the Stock Exchange Automated Trading System (SEATS), ASX futures and the company announcement service. All daily stock exchange activity is available in different electronic data feeds that the ASX calls "signals." In our work, we have used announcements from the ASX's Signal G, which provides subscribers with company announcements issued by companies or the ASX in accordance with listing rules.

Section 2 of this paper describes the Signal G data set. Section 3 describes the different machine learning techniques and the categorization framework that we have used to perform these types of categorization tasks. Section 4 describes the quantitative results and section 5 concludes.

2 Data Description

In this paper we assess performance on two tasks: "report type" and "market sensitivity" categorization. Report type is "a code to categorize company announcements" [1] and may take one of 144 values like "annual report" or "takeover announcement." Market sensitivity is a boolean category indicating whether an announcement contains information that may influence trading in the issuing company. This allows users of this data to select which announcements are critical.

Currently, both kinds of category assignments on the Signal G documents are made manually by the ASX. Table 1 shows the distribution of docu-

Proceedings of the 7th Australasian Document Computing Symposium,
Sydney, Australia, December 16, 2002.

	Sensitive	Nonsensitive	Total
Training	32458	63067	95525
Test	14072	27033	41405
Total	46530	90100	136630

Table 1: Sensitivity category distribution in Signal G

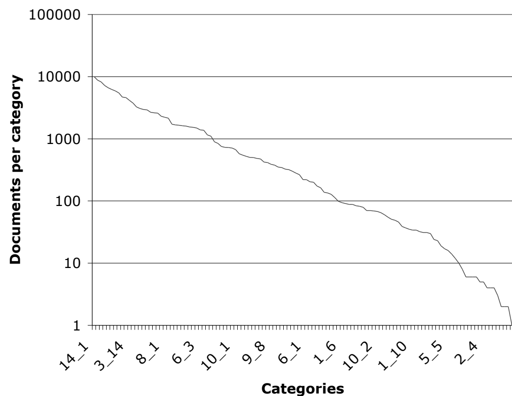

Figure 1: Report type category distribution in Signal G

ments with respect to the sensitivity category, as well as our split between training and testing documents. Figure 1 shows the report type category distribution.

Note that the report type labels are highly skewed across categories. The most common category contains 10,064 documents, while ten categories have fewer than five documents each.

Signal G is available to the public, member organizations, and information vendors. Our data set was supplied by the Capital Markets Collaborative Research Centre. For future groups who receive permission from the CMCRC to work with this data, we have converted it into an XML format.

3 Methods

We have compared three machine learning methods that have provided some of the best performances in other classification tasks [16]: Support Vector Machines (SVM) [12] [6], Naïve Bayes (NB) [7], and Neural Networks (NN) [5] [4]. It is not possible to describe them thoroughly in this article, so we will only summarize those issues that might be required to reproduce the results. Our Naïve Bayes and SVM implementations are described in [15].

Because of the inability of our SVM implementation to handle the size of our training set, we had to only train on 8000 documents at a time in order to finish the experiments in a reasonable amount of time. The SVM was not able to perform the report type task, only the sensitivity task. Future work may dramatically improve training speed by using recently developed training optimizations in the literature, enabling us to train on a larger number of documents and presumably to improve our correctness as a result.

Our Neural Network architecture [5] [4] used a backpropagation algorithm that minimizes quadratic error. The input layer has as many units (neurons) as document features retained. The number of hidden units is determined by optimising the performance on a cross-validation set. There is one output unit for each possible category, with activation values between 0 and 1. For each document, the classifier will assign any category whose output unit is greater than 0.5. In our experiment, we used 3-fold cross-validation and averaged the weights of the three resultant neural networks.

Each categorizer was trained on the 95,525 training documents for each task, report type and sensitivity. For the Neural Network categorizer, 13,711 training documents were set aside from the training set to function as a validation set when tuning the weights of the network. The trained categorizers were then evaluated on the 41,405 test documents. Since documents in the test set have not been used to adjust the parameters of the classifier, it is normally assumed that the performance on new data would be similar.

Table 2 summarizes the general steps followed in the preparation of the experiments. TF/IDF weights are given in the notation followed by [11]. Our experimental process was as follows:

1. Linguistic dimensionality reduction: A list of stopwords [10] was removed from the document collection and the Porter stemming algorithm was applied [8].

2. Statistical dimensionality reduction: Chi Squared or Document Frequency criteria were employed to reduce the feature vector dimensionality [8] [13].

3. Vectorization and weighting: The resulting documents were represented as vectors, using TF/IDF weighting [11] [17].

4. Architecture: The selected terms were used as input features to the classifier. Some of the algorithms allow several architectures, and the best algorithm was chosen by optimising the results on a cross-validation set.

5. Training: We generated a cross-validation set randomly. These documents were set aside and the Neural Network was trained on the remaining ones.

4 Results

In evaluating our classifiers on our data set, we use the common statistical measures *precision*, *re-*

	Stopwords	Stemming	Feature Reduction	TF/IDF	Architecture
NN	SMART	Porter	χ^2	tfc	1000 features, 50 hidden units
NB	SMART	Porter	DF	tfx	1000 features
SVM	SMART	Porter	DF	tfx	1000 features, linear kernel

Table 2: Comparative description of algorithms used

	Micro			Macro		
	p	r	F_1	p	r	F_1
NN	0.89	0.89	0.89	0.88	0.88	0.88
NB	0.83	0.84	0.83	0.90	0.90	0.90
SVM	0.82	0.82	0.82	0.80	0.79	0.80

Table 3: Performance for the market sensitivity task

	Micro			Macro		
	p	r	F_1	p	r	F_1
NN	0.87	0.71	0.78	0.45	0.34	0.37
NB	0.62	0.67	0.64	0.46	0.61	0.46

Table 4: Performance for the report type task

call, and F_1. [16] [14] When dealing with multiple classes there are two possible ways of averaging these measures, *macro-averaging* and *micro-averaging*. The macro-average weights equally all the classes, regardless of how many documents they contain. The micro-average weights equally all the documents, thus biasing toward the performance on common classes. Since different learning algorithms will perform differently on common and rare categories, both micro-averaged and macro-averaged scores are typically reported to evaluate performance.

It is important to note that the performance results are based on comparing the automatic categorization of each document with the tagging of human experts at the ASX. The manual classification is a subjective decision process affected by the ASX's legal liabilities and the normal human classification disagreements. It has been shown in various studies that there could be considerable variation in the inter-indexer agreement [3] [2]. For example, in a Reuters news collection correction rates averaged 5.16% [9] with some editors being corrected up to 77% of the time. Similar disagreements can be expected in the ASX's assignments on the Signal G corpus. In the light of these disagreements we can imagine that there might be a limit to the performance that can be obtained by automatic categorization.

Figure 2 shows a histogram of classes and documents for different performance ranges using the Naïve Bayes classifier. It shows how most categories that do well have large number of documents, except for some that have very few documents (fewer than 5). This shows the well-known result that the machine learning algorithms such

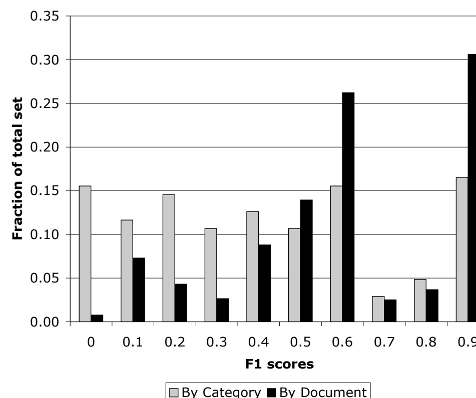

Figure 2: Histogram of report type results for different performance ranges

as Naïve Bayes perform better on well-populated categories. Similar results can be obtained for the other classifiers.

5 Conclusion

In this paper we have applied several machine learning techniques (Neural Networks, Naïve Bayes, and Support Vector Machines) to the categorization of announcements of companies publicly traded by the ASX. Two tasks were evaluated: the categorization of documents as sensitive or not, and the categorization in one of the 144 report types defined by ASX. The results show that is possible to obtain classifiers with more than 88% precision and recall on the sensitivity task and 86% precision, 74% recall on the report type task.

The results are somewhat better than the ones obtained by several researchers working on the Reuters news cable database [5] [4]. This database has fewer categories (90) than the report type task but also fewer documents (10,000). Although it is risky to try to extrapolate the results, we believe that due to the similarity in the documents, other financial databases with documents in English should also have similar performance. Future work includes testing adding statistical feature selection to the classification framework, and improving the efficiency of the algorithms so they can be used for even larger data sets.

The excellent performance shows the possibility to use these classifiers in commercial applications

for both tasks, sensitivity detection and report type categorization.

Acknowledgements

The authors gratefully acknowledge financial support from the Capital Markets Collaborative Research Centre and the University of Sydney.

References

[1] Australian Stock Exchange web site. http://www.asx.com.au/, 2002.

[2] Thorsten Brants. Inter-annotator agreement for a german newspaper corpus. In *2nd International Conference on Language Resources & Evaluation*, 2000.

[3] R. Bruce and J. Wiebe. Word sense distinguishability and inter-coder agreement. In *3rd Conference on Empirical Methods in Natural Language Processing (EMNLP-98)*, Granada, Spain, June 1998. Association for Computational Linguistics SIGDAT.

[4] Rafael A. Calvo. Classifying financial news with neural networks. In *6th Australalasian Document Symposium*, page 6, December 2001.

[5] Rafael A. Calvo and H. A. Ceccatto. Intelligent document classification. *Intelligent Data Analysis*, Volume 4, Number 5, 2000.

[6] Thorsten Joachims. Transductive inference for text classification using support vector machines. In Ivan Bratko and Saso Dzeroski (editors), *Proceedings of ICML-99, 16th International Conference on Machine Learning*, pages 200–209, Bled, SL, 1999. Morgan Kaufmann Publishers, San Francisco, US.

[7] David D. Lewis. Naive (Bayes) at forty: The independence assumption in information retrieval. In Claire Nédellec and Céline Rouveirol (editors), *Proceedings of ECML-98, 10th European Conference on Machine Learning*, number 1398, pages 4–15, Chemnitz, DE, 1998. Springer Verlag, Heidelberg, DE.

[8] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.

[9] Tony G. Rose, Mark Stevenson and Miles Whitehead. The reuters corpus volume 1 - from yesterday's news to tomorrow's language resources. In *3rd International Conference on Language Resources and Evaluation*, page 7, May 2002.

[10] Gerard Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley, Reading, Pennsylvania, 1989.

[11] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management: an International Journal*, Volume 24, Number 5, pages 513–523, 1988.

[12] Bernhard Schölkopf, Christopher J.C. Burges and Alexander J. Smola (editors). *Advances in Kernel Methods – Support Vector Learning*. MIT Press, 1999.

[13] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys (CSUR)*, Volume 34, Number 1, pages 1–47, 2002.

[14] C. J. Van Rijsbergen. *Information Retrieval, 2nd edition*. Butterworths, 2 edition, 1979.

[15] Ken Williams and Rafael A. Calvo. A framework for text categorization. *7th Australasian Document Computing Symposium*, 2002.

[16] Y. Yang and X. Liu. A re-examination of text categorization methods. In *22nd Annual International SIGIR*, pages 42–49, Berkley, August 1999.

[17] Yiming Yang and Jan O. Pedersen. A comparative study on feature selection in text categorization. In Douglas H. Fisher (editor), *Proceedings of ICML-97, 14th International Conference on Machine Learning*, pages 412–420, Nashville, US, 1997. Morgan Kaufmann Publishers, San Francisco, US.

Appendix C: Automatic Categorization of Questions for a Mathematics Education Service

Ken Williams Rafael A. Calvo
David Bell
Web Engineering Group
The University of Sydney
Bldg J03, Sydney NSW 2006

{*kenw,rafa*}*@ee.usyd.edu.au*, *dave@student.usyd.edu.au*

Abstract. This paper describes a new approach to managing a stream of questions about mathematics by integrating a text categorization framework into a relational database management system. The corpus studied is based on unstructured submissions to an ask-an-expert service in learning mathematics. The classification system has been tested using a Naïve Bayes learner built into the framework. The performance results of the classifier are also discussed. The framework was integrated into a PostgreSQL database through the use of procedural trigger functions.

1 Introduction

Ask-an-expert services are becoming more common, spanning from standard customer relationship management to discussion forums in a particular discipline. In general, these online services are supported by domain experts who attempt to answer questions posted via email or web forms. Since these experts often have a single subdomain of expertise it is very helpful if they have only to read questions that relate to this subdomain. This can be done by organizing the service in such a way that users are encouraged to post their question in the appropriate area. However, this approach is not always successful as often the user will either ignore the organization scheme or not know to which area their question belongs.

These problems are common within a number of domains. Our test was performed on messages sent to a mathematics ask-an-expert service for students and teachers.[6] The issues discussed also apply to other similar systems such as customer relationship management (CRM) and e-learning systems in general. These systems can use an automatic text categorization framework to categorize the questions into the experts' areas of interest, or into the appropriate customer support mailbox.

The downside of an automatic categorization approach is that integrating such functionality into existing systems can be very complex, and often involves an in depth understanding

Proceedings of the 11th International Conference on Artificial Intelligence in Education, Sydney, Australia, July 20-24, 2003. Submitted January 7, 2003, accepted February 24, 2003.

```
Implicit Differentiation

Find the slope of the tangent at the point
(3,4) on the circle x^2 + y^2 = 25.

My answer: I guess we would need to put it
in the y = mx + b form.

Thanks for any help,
Scott
```

Figure 1: An example document from the Dr. Math corpus

of text categorization techniques. Also, the content is normally stored in systems with a relational database in the backend, as is the case for most content and learning management systems. By building the categorizer into the database, the categorization framework[11] can be made invisible to the users and is thus more attractive to the average system administrator or application developer. Also, application developers, do not have to re-implement the classification software. They only need a machine learning professional to assist in training the classifier, and once trained it can then be reused for different applications.

The applications of information retrieval have been well studied since the 1980s, as discussed by Salton [9, 8], and many of these methodologies have been integrated into commercial database management systems that have free text search capabilities. However, this integration does not seem to have penetrated the text categorization domain.

Section 2 of the paper discusses the data set that was used to test the system. Section 3 discusses the text categorization framework and the extensions made to it, including the implementation within the database management system. Section 4 discusses the quantitative results of the testing process and Section 5 concludes.

2 Dr. Math Corpus

For the evaluation of our system we have tested the performance of the categorization system over a set of unstructured, informal documents from the Ask Dr. Math service.[6] These documents are mostly written by students between the ages of 6 and 18, though question submissions can come from any member of the general public. The documents vary in length from a single sentence to several paragraphs. In addition to this, many examples contain symbols and diagrams, making linguistic analysis very difficult. The Ask Dr. Math service has about 300 volunteers (about 30-40 of which may be active in any given month), dealing with hundreds of questions a day. The volunteers have expertise in different areas of math, and the site has won a number of awards for its useful service. Figure 1 shows an example submission to the service.

The filtering of questions is a major element of the Ask Dr. Math question answering process. The service may receive about 7000 questions a month, about half of which are eventually answered. The unanswered questions may be duplicate submissions, messages of thanks, inappropriate questions, or other messages that don't require a response. There also may be some legitimate mathematics questions that go unanswered, simply because

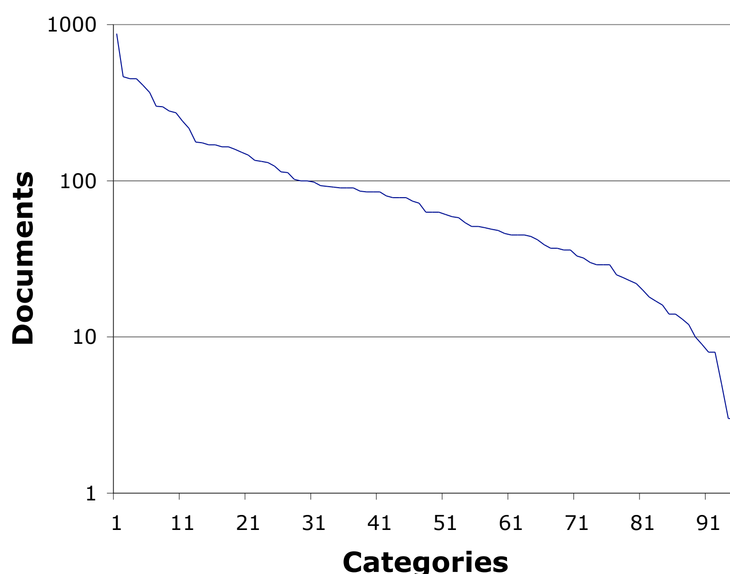

Figure 2: Category distribution for the 95 Dr. Math categories

the service is not fee-based for either the students or the experts, and thus can make no guarantee that any particular question will be answered. The experts are currently responsible for choosing their own questions to answer.

The Dr. Math corpus we used contains 6632 documents and was split into a training set of 5305 documents and a testing set of 1327 documents. There are 95 categories in the corpus, and the average number of documents in each category is 107.15. The categories are structured in a 2-level hierarchy ("level" and "topic"), with a total of four levels (elementary school, middle school, high school, and college) and 62 topics (for example, "geometry" or "statistics"). The most popular category, high school-level geometry, contains 877 documents, and the least popular category, elementary-level golden ratio, contains only 3 documents. Each document may be a member of more than one category, and the average number of categories per document is 1.53. Figure 2 shows the distribution of categories throughout the corpus.

It may be important to note that our corpus was drawn from the public archive of answered questions, not directly from the stream of incoming questions. Because the archiving process is fairly intensive and not all questions are chosen for archiving, our corpus may therefore differ significantly from the incoming question stream. For example, none of the kinds of unanswered questions mentioned earlier are represented in the archive. Because of this difference, it is difficult to extrapolate our experiments to performance on the incoming question stream. However, because the incoming question stream is uncategorized, obtaining a large enough number of categorized questions for our investigation necessitated drawing them from the archived questions.

```
INSERT INTO documents
 (name, content, categories)
VALUES
 ('my name',
  'my content',
  categorize('my name',
             'my content',
             'documents'));
```

Figure 3: Example document insertion statement with categorization

3 Categorization Framework

Object Oriented Application Frameworks (OOAF) are software engineering artifacts that improve reusability of design and implementation [4, 5].

The framework used in this project was designed to provide a consistent way to build document categorization systems.[11] It allows the developer to focus on the quality of the more critical and complex modules by allowing reuse of common, simple base modules. The framework has implementations of k-Nearest-Neighbor (kNN), Naïve Bayes (NB), Support Vector Machine (SVM), and Decision Tree (DT) classifiers [12, 10]. Other methods such as Neural Network [3, 2] classifiers are under development.

The framework architecture allows extensions to be built by subclassing its main classes [11]. Class inheritance contributes to code reuse and quality. In this project we extended the framework by adding an alternative Collection class to allow for the data to be read directly from a database instead of from a file system. Having a classifier that uses the data directly from the database streamlines the management of questions and answers in this type of system. In fact, it allows many content or learning management systems to natively use automatic classification features. The framework also provides statistical analysis of experimental results, and produced the performance measures discussed in Section 4.

The framework's architecture and language choice enabled us to easily build the framework into postgreSQL through postgreSQL's PL/Perl and PL/perlU support. This support allows the creation of procedural language functions through the use of the "CREATE FUNCTION" SQL command. Using this support and the PL/perlU language we were able to build a "launching" function that invoked the categorization framework on the document to be classified. This means that the only command necessary to categorize a document is a basic insert statement with a function call in place of a value for the category of the document, as shown in Figure 3. This statement can be further simplified through the creation of a pl/psql trigger function which fires automatically on insertion and passes the necessary values to the `categorize()` function.

If the categorization is to take place within a database, where categorized documents are often going to be appended to the learning set, a learning algorithm which has very little training overhead is ideal. This avoids the need to retrain a categorizer after each document insertion. Two such algorithms are the Naïve Bayes algorithm (NB) and the K-Nearest Neighbor algorithm (kNN).

The training phase in NB consists only of counting term frequencies in each document and using them to calculate conditional probabilities between terms and categories. These

	MaP	MaR	MaF_1	MiF_1	Error
NB	0.246	0.226	0.226	0.361	0.022
kNN	0.211	0.186	0.179	0.257	0.025
Baseline	0.019	0.018	0.018	0.042	0.031

Table 1: Macro- and Micro-averaged performance scores.

probabilities are then consulted when categorizing a new document, with conditional probabilities for each term being multiplied to find the probability that a given document belongs to a certain category.

kNN in its basic form has essentially no training phase. Each document is represented as an n-dimensional vector, where n is the number of unique terms in the training set. When a new document is to be classified, it is compared to the vectors of the documents in the training set. The k training vectors which are closest to the test vector are found (with distance defined as the cosine of the angle between any two vectors), and the most prevalent category or categories amongst these is assigned to the new document.

In our testing, we have found that NB is a more accurate categorizer than kNN. The rest of this paper will focus on the NB experiment and results.

Since the categorization performance is determined only by the classification framework, all these methods should behave the same inside the database as they do outside the database. What the integration into the database does is to make the functionalities of the framework available as procedures in the SQL language. Since relational databases can be designed using an object oriented methodology [1, 7], by integrating it in this way, the classification task (and framework) can also be designed into larger OO systems.

4 Method and Results

The 5305 training documents in the Dr. Math corpus were loaded into a database table named "documents." This table consisted of 3 columns: name, content and categories. The testing set was then inserted into the database, one document at a time using a statement similar to that in Figure 3. A SELECT statement was then used to compare the assigned and actual categories of each document. Through this comparison the performance of the categorization in terms of precision and recall was measured.[1] The precision and recall were then used to calculate the F_1 measure [2, 10].

The results in Table 1 show the performance of the categorization algorithms. The precision, recall, and F_1 scores can be computed using macro-averaging, which gives equal weight to each category, or micro-averaging, which gives equal weight to each document. [12, 10] For the kNN algorithm, we used a k value of 15 and a categorization threshold of 0.12, which seemed to perform the best in our investigations. We also include in Table 1 the results of a baseline classifier, which assigns categories at random to each test document, weighting the random generator by the frequency of categories in the training set.

Next, we turned our attention to ways of improving performance on the test set. As mentioned in Section 2, each category name is a combination of two components, a level and a

[1] Recall is the proportion of the target items that the system selected, i.e. tp/(tp+fn). Precision is the proportion of selected items the system got right, i.e. tp/(tp+fp).

Task	MaP	MaR	MaF_1	MiF_1	Error
Level	0.524	0.626	0.570	0.671	0.223
Topic	0.339	0.314	0.313	0.440	0.026
Both	0.187	0.181	0.166	0.223	0.035

Table 2: Performance of Naïve Bayes classifier on subtasks.

Task	MaP	MaR	MaF_1	MiF_1	Error
Level	0.326	0.319	0.322	0.468	0.328
Topic	0.029	0.027	0.027	0.067	0.041
Both	0.015	0.010	0.011	0.035	0.027

Table 3: Performance of baseline classifier on subtasks.

topic. This suggests that separate categorizers could be trained to recognize the two components separately, perhaps with more success than a single categorizer may have on the two components together.

To test this hypothesis, we created three new categorization tasks: one that categorizes by level alone, one that categorizes by topic alone, and one that uses the separate topic and level categorizers to assign a combined category. In this combined process, each assigned level was combined with each assigned topic, any nonexistent categories (such as "calculus.elem" or "addition.college") were filtered out, and all remaining categories were assigned to the given document. This process was performed using the Naïve Bayes and baseline categorizers described above. Table 2 shows the performance for the Naïve Bayes categorizer and Table 3 shows the baseline categorizer for comparison.

Comparing the combined task to the NB results in Table 1, we see that separating the categorization task into two subtasks adversely affected the overall performance on the combined task. The performance for the level task seems good, but comparing it with the baseline categorizer shows that it may not be significantly better than random guessing. This is probably due to the small number of categories. However, the performance on the topic task is noteworthy, because it is so far above both the baseline categorizer and the original NB categorizer. In addition, the topic assignment may be more valuable in this application than the level assignment, because while most students will be able to indicate their own age or grade level when asking a question, they may not be able to place their own question in an appropriate category.

Because the math topics used in this experiment were generated from the category names, we ended up with some duplication that may have decreased performance. For example, there are categories called "probability.high," "statistics.high," and "prob.stats.middle." This means that the combined list of extracted math topics includes "probability," "statistics," and "prob.stats." The exact effect of this on the categorizers is unknown, but we might expect these overlapping categories to confuse the categorizer. A service such as this one may therefore benefit from using more consistent category names.

5 Conclusion

We have described a system that integrates a categorization framework into a relational database. The results show it is possible to integrate categorization techniques into the relational databases used by learning and content management systems.

Two categorization algorithms were applied to the task of classifying messages sent to an educational ask-an-expert service. A Naïve Bayes classifier outperformed the kNearest Neighbour classifier and was reasonably successful at categorizing the messages. Future work includes testing classifiers that use other machine learning models such as Support Vector Machines and Neural Networks. The classification performance of Naïve Bayes was also measured using the 2-level hierarchy of the corpus. In this case, the highest success rate was found in classifying messages by math topic, instead of school level. This task could be useful on the unstructured data sent to ask-an-expert services such as the one we discussed.

Acknowledgements

D. Bell gratefully acknowledges support from the Capital Markets Collaborative Research Centre and the University of Sydney. The authors also acknowledge The Math Forum for giving permission to use and discuss their data.

References

[1] Michael R. Blaha, William J. Premerlani, and James E. Rumbaugh. Relational database design using an object-oriented methodology. *Communications of the ACM*, 31(4):414–427, 1988.

[2] Rafael A. Calvo. Classifying financial news with neural networks. In *6th Australasian Document Symposium*, page 6, December 2001.

[3] Rafael A. Calvo and H. A. Ceccatto. Intelligent document classification. *Intelligent Data Analysis*, 4(5), 2000.

[4] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[5] Mohamed Fayad and Douglas C. Schmidt, editors. *Building Application Frameworks*. John Wiley & Sons, 1999.

[6] The Math Forum. Ask Dr. Math web site. http://www.mathforum.org/dr.math/.

[7] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented modeling and design*. Prentice-Hall, Inc., 1991.

[8] Gerald Salton. Developments in automatic text retrieval. *Science*, (253):974–979, 1991.

[9] Gerard Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley, Reading, Pennsylvania, 1989.

[10] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys (CSUR)*, 34(1):1–47, 2002.

[11] Ken Williams and Rafael A. Calvo. A framework for text categorization. *7th Australasian Document Computing Symposium*, 2002.

[12] Yiming Yang and X. Liu. A re-examination of text categorization methods. In *22nd Annual International SIGIR*, pages 42–49, Berkley, August 1999.