Paper Categorization Using Naive Bayes

by

Man Cui
B.Sc., University of Victoria, 2007

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Paper Categorization Using Naive Bayes

by

Man Cui
B.Sc., University of Victoria, 2007

Supervisory Committee

Dr. Bill Wadge, Supervisor
(Department of Computer Science)

Dr. Bruce Kapron, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Bill Wadge, Supervisor
(Department of Computer Science)

Dr. Bruce Kapron, Departmental Member
(Department of Computer Science)

ABSTRACT

Literature survey is a time-consuming process as researchers spend a lot of time in searching the papers of interest. While search engines can be useful in finding papers that contain a certain set of keywords, one still has to go through these papers in order to decide whether they are of interest. On the other hand, one can quickly decide which papers are of interest if each one of them is labelled with a category. The process of labelling each paper with a category is termed paper categorization, an instance of a more general problem called text classification. In this thesis, we presented a text classifier called Iris that makes use of the popular Naive Bayes algorithm. With Iris, we were able to (1) evaluate Naive Bayes using a number of popular datasets, (2) propose a GUI for assisting users with document categorization and searching, and (3) demonstrate how the GUI can be utilized for paper categorization and searching.

Contents

Supervisory Committee ii

Abstract iii

Table of Contents iv

List of Tables viii

List of Figures ix

Acknowledgements xi

1 Introduction 1
- 1.1 Contributions . . . 2
- 1.2 Thesis Organization . . . 3

2 Background on Naive Bayes 4
- 2.1 Naive Bayes Text Classification . . . 5
- 2.2 Multi-Variate Bernoulli Model . . . 5
  - 2.2.1 Training . . . 5
  - 2.2.2 Applying . . . 7
- 2.3 Multinomial Model . . . 8
  - 2.3.1 Training . . . 8
  - 2.3.2 Applying . . . 10
- 2.4 Underflow Problem . . . 11
  - 2.4.1 Multi-Variate Bernoulli Model . . . 11
  - 2.4.2 Multinomial Model . . . 12
- 2.5 Feature Selection . . . 13

3 Iris: A Naive Bayes Text Classifier 15

3.1 Train Iris . . . 15
3.1.1 Training Set . . . 16
3.1.2 Commands . . . 16
3.2 Apply Iris . . . 18
3.2.1 Test Set . . . 18
3.2.2 Commands . . . 19
3.3 Search Documents by GUI . . . 21
3.3.1 Command . . . 21
3.3.2 Approach . . . 22

4 Iris Design and Implementation 26
4.1 The Database . . . 26
4.1.1 The `word` Table . . . 26
4.1.2 The `document` Table . . . 28
4.1.3 The `category` Table . . . 28
4.1.4 The `document_category` Table . . . 28
4.1.5 The `inverted_index` Table . . . 28
4.1.6 The `word_category_probability` Table . . . 29
4.2 The `util` Module . . . 29
4.2.1 The `DB` Class . . . 29
4.2.2 The `get_word_frequencies` Function . . . 31
4.3 The Commands . . . 32
4.3.1 frequency.py . . . 32
4.3.2 mutual_info.py . . . 32
4.3.3 probability.py . . . 33
4.3.4 categorize.py . . . 34
4.4 The GUI . . . 34

5 Performance Evaluation 37
5.1 Datasets . . . 37
5.1.1 20 Newsgroups . . . 37
5.1.2 Reuters-21578 . . . 38
5.1.3 WebKB . . . 40
5.2 Data Processing . . . 41
5.3 Training Results . . . 45

5.4 Test Results for Stemmed 45
5.5 Test Results for All Terms 49
5.6 Discussion 49

6 Paper Categorization **52**
6.1 Collecting Papers 52
6.1.1 Training Set 53
6.1.2 Test Set 53
6.2 Setup 53
6.2.1 Training Iris 55
6.2.2 Performance Evaluation 55
6.2.3 Applying Iris 55
6.3 Searching Papers 55
6.3.1 Searching by a Query String 57
6.3.2 Searching by Categories 57
6.3.3 Searching by a Query String and Categories 59
6.4 Discussion 59

7 Related Work **60**
7.1 Variants of Bayesian Text Classifiers 60
7.1.1 Multi-Variate Bernoulli Model 61
7.1.2 Multinomial Model 61
7.1.3 Restricted Bayesian Network 61
7.1.4 Hierarhical Classification 62
7.1.5 Multi-Label Classification 62
7.2 Feature Selection 63
7.3 Evaluation Metrics 63
7.3.1 Accuracy 63
7.3.2 Precision and Recall 63

8 Conclusions **65**
8.1 Contributions 65
8.2 Future Work 67
8.2.1 Hierarchical Classification 67
8.2.2 Multi-Label Classification 67

A Database Schema **68**

B Python Modules and Iris Commands **70**

B.1 util.py 70

B.2 frequency.py 79

B.3 mutual_info.py 81

B.4 probability.py 83

B.5 categorize.py 85

Bibliography **89**

List of Tables

Table 2.1 Internal representation of the training set 6
Table 2.2 P(Word|Category) values calculated by Laplace smoothing . . . 6
Table 2.3 P(Category) values calculated from Table 2.1 7
Table 2.4 Internal representation of a test document 8
Table 2.5 Internal representation of the training set 9
Table 2.6 P(Word|Category) values calculated by Laplace smoothing . . . 9
Table 2.7 Internal representation of a test document 10

Table 5.1 Topics for 20 Newsgroups 38
Table 5.2 Categories for WebKB 40
Table 5.3 Training set versus test set for 20 Newsgroups 42
Table 5.4 Training set versus test set for Reuters 21578 42
Table 5.5 Training set versus test set for WebKB 43

List of Figures

Figure 3.1 Example directory structure for training set 16
Figure 3.2 Example training set 17
Figure 3.3 Example directory structure for test set 18
Figure 3.4 Example test set 19
Figure 3.5 Example summary 20
Figure 3.6 Iris GUI startup 22
Figure 3.7 A progress bar for text categorization 23
Figure 3.8 Screenshot for searching by query string 24
Figure 3.9 Screenshot for searching by category 24
Figure 3.10 Screenshot for searching by query string and category 25
Figure 3.11 Screenshot for viewing document 25

Figure 4.1 E/R diagram: rectangles represent entities, diamonds represent relationships, and ovals represent attributes. Arrow means one-to-many, otherwise many-to-many. 27
Figure 4.2 Call graph of `gui.py` 36

Figure 5.1 Training time (in seconds) 44
Figure 5.2 Average word count per document 44
Figure 5.3 Test results for `20news_stemmed` 46
Figure 5.4 Test results for `reuters_stemmed` 47
Figure 5.5 Test results for `webkb_stemmed` 48
Figure 5.6 Test results for `20news_all_terms` 50
Figure 5.7 Test results for `reuters_all_terms` 51

Figure 6.1 Hierarchical categories extracted from ACM CCS 54
Figure 6.2 Test results for `paper_abstracts` 56
Figure 6.3 Screenshot for searching by query string 57
Figure 6.4 Screenshot for searching by category 58

Figure 6.5 Screenshot for searching by query string and category 58

Figure 7.1 Bayesian networks 61
Figure 7.2 Category hierarchy 62

ACKNOWLEDGEMENTS

I am happy to present this thesis to my supervisor, Professor Bill Wadge, who provided me with directions on how to proceed with my research and allowed me to work part-time in James Evans and Associates, a Victoria-based company where I gained valuable experience in database programming and web application development. I would also like to thank my husband and parents for their patience and constant encouragement. The thesis could not have been completed without their ongoing support.

Chapter 1

Introduction

Researchers spend a lot of time in literature surveys as they attempt to obtain a thorough understanding of various research topics. A literature survey usually starts with a few papers that a researcher may find relevant to the topic which he or she is investigating. Obviously, these few papers alone do not provide enough coverage of the topic under investigation. The researcher therefore collects more papers by going through the references of the aforementioned papers. This process continues until a desirable number of papers are collected.

Thanks to the World Wide Web (WWW), most papers nowadays are available online. A researcher can use a search engine of his or her choice to locate the papers of interest. With search engines, there are two approaches of paper searching. The first approach involves typing the keywords that the papers of a particular topic are likely to contain and is therefore referred to as *search by keywords*. This is used at the start of a literature survey as the researcher has no relevant papers at hand. The second approach involves typing the name of a paper and is therefore referred to as *search by name*. This is used when the researcher goes through the references of the relevant papers. In both cases, the papers returned by the search engines may or may not relate to the topic which the researcher is interested in. As a result, the researcher has to decide the relevance of each paper by themselves, a potentially time-consuming task.

This problem can easily be solved by the use of Iris, a prototype that we developed for demonstrating the power of paper categorization. Iris is a text classifier that offers a command line interface as well as a Graphical User Interface (GUI). The command line interface is used for training the text classifier and testing its accuracy and speed against a dataset of choice. The GUI, on the other hand, allows the user to apply

the text classifier to documents of unknown categories and search for the documents of interest.

The text classification algorithm used by Iris is called Naive Bayes, a popular algorithm in the text classification literature. There are two reasons that we chose Naive Bayes over other algorithms. According to Cardoso-Cachopo [1], Naive Bayes achieves high accuracy in a number of datasets. It is also very fast, possibly because its algorithm is relatively simple.

With Iris GUI, the researcher is presented with a query field and a number of checkboxes, one for each category. These two widgets together can be used to search papers in three different modes:

- *searching by a query string*: returns all papers that contain the query string, regardless of the paper categories. With this approach, the researcher enters the query in the query field and checks all the categories.
- *searching by categories*: returns all papers that are of the specified categories. With this approach, the researcher leaves the query field blank and checks the categories of interest.
- *searching by a query string and categories*: returns all papers that contain the query string and are of the specified categories. With this approach, the researcher enters the query in the query field and checks the categories of interest.

In all three modes, the matched papers are organized according to their respective categories.

1.1 Contributions

The contributions of this thesis are listed as follows:

- Introduced a prototype that facilitates the evaluation of Naive Bayes against datasets of choice.
- Presented the results of running Naive Bayes against a number of popular datasets.
- Proposed a GUI for assisting users with categorizing documents of unknown categories and locating the documents of interest.
- Demonstrated the steps of paper categorization and searching through GUI.

1.2 Thesis Organization

Below is a brief description for each of the chapters that follows. Chapter 2 explains what Naive Bayes is and the calculation steps involved in training and applying. Chapter 3 and 4 describe the Iris prototype in terms of usage, design, and implementation. Chapter 5 provides the result of running Iris against a number of popular datasets. Chapter 6 presents a case study on paper categorization using Iris. Chapter 7 summaries the related work found in the text classification literature. Finally, Chapter 8 concludes the work presented in this thesis. In addition, the Iris implemention is detailed in Appendix A and B.

Chapter 2

Background on Naive Bayes

Naive Bayes is based on the Bayes' rule, an equation for computing conditional probabilities. A conditional probability is defined as the probability of an event given that another event has occurred. For example, consider two events A and B with $P(B) > 0$. The conditional probability, denoted as $P(A|B)$, is the probability of event A given that event B has occurred. According to the Bayes' rule, $P(A|B)$ is calculated as:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.1}$$

$P(B)$ is referred to as a priori probability because it is the probability of event B *before* the occurrence of event A. $P(B|A)$ is referred to as a posterior probability because it is the probability of event B *after* the occurrence of event A.

A generalization of the above equation is to replace B with a set of events. Then the new equation becomes:

$$P(A|B_1, B_2, ..., B_n) = \frac{P(B_1, B_2, ..., B_n|A)P(A)}{P(B_1, B_2, ..., B_n)} \tag{2.2}$$

Since $P(B_1, B_2, ..., B_n|A)$ is difficult to compute in practice, Naive Bayes assumes that the events, $B_1, B_2, ..., B_n$ are independent of each other. With this simplifying assumption, the above equation can be rewritten as follows:

$$P(A|B_1, B_2, ..., B_n) = \frac{P(B_1|A)P(B_2|A)...P(B_n|A)P(A)}{P(B_1)P(B_2)...P(B_n)} \tag{2.3}$$

2.1 Naive Bayes Text Classification

The problem of text classification can be described as: given a document containing a list of words, what is the category of this document? In order to answer this question, we first need to train a text classifier with a set of documents of known categories, hereafter referred to as the training set. This involves extracting the following items from the training set: the set of document categories, the set of words, and how the words are distributed among the documents. We can then use this information to compute the score for each category. The category with the highest score is—as determined by the text classifier—the document's category.

Let the categories be $C_1, C_2, ..., C_m$, the score of a document belonging to category C_i for $1 \leq i \leq m$ is denoted as $score(C_i)$. In the context of Naive Bayes text classification, $score(C_i)$ is calculated according to the event model used. There are two commonly used event models for Naive Bayes text classifiers: the multi-variate Bernoulli model and the multinomial model. The difference between the two lies in whether the word frequencies are used in the calculation of the probabilities. With the multi-variate Bernoulli model, a document is represented as a list of 1's and 0's where a 1 indicates the presence of a word whereas a 0 indicates otherwise. With the multinomial model, a document is represented as a list of word frequencies.

2.2 Multi-Variate Bernoulli Model

2.2.1 Training

The first step in training a Naive Bayes text classifier is to extract the words from the training set. Table 2.1 shows the internal representation of an example training set. The first column shows the set of document identifiers. The second column shows the document categories. The rest of the columns show the presence and absence of the words in each document. For example, the word `screen` is present in document D1 but absent in document D2.

The next step is to calculate $P(\text{Word}|\text{Category})$ for all combinations of words and categories. Given a word W and a category C,

$$P_1(W|C) = \frac{N_{WC}}{N_C} \tag{2.4}$$

Training Set	Category	*screen*	*coverage*	*disk*	*processor*	*wi-fi*	*signal*
D1	*cell*	1	1	0	1	1	1
D2	*cell*	0	1	0	0	1	1
D3	*cell*	0	1	0	0	1	1
D4	*laptop*	1	0	1	1	1	0
D5	*laptop*	1	0	1	1	1	0
D6	*laptop*	1	0	1	1	1	0

Table 2.1: Internal representation of the training set

	screen	*coverage*	*disk*	*processor*	*wi-fi*	*signal*
cell	0.4000	0.8000	0.2000	0.4000	0.8000	0.8000
laptop	0.8000	0.2000	0.8000	0.8000	0.8000	0.2000

Table 2.2: P(Word|Category) values calculated by Laplace smoothing

- N_{WC} is the number of documents of category C containing the word W.
- N_C is the number of documents of category C.

Using Table 2.1 as an example, the number of documents of category *laptop* containing the word *screen* is $1 + 1 + 1 = 3$. The number of documents of category *laptop* is 3 (i.e. D4, D5, and D6). Therefore, $P(screen|laptop)$ is $3/3 \approx 1.0000$. As another example, the number of documents of category *cell* containing the word *coverage* is $1 + 1 + 1 = 3$. The number of documents of category *cell* is 3 (i.e. D1, D2, and D3). Therefore, $P(coverage|cell)$ is $3/3 \approx 1.0000$.

However, there is a problem with this approach. Given a training set, sometimes a word never appears in documents of a particular category. Using Table 2.1 as an example, the word *disk* never appears in documents of category *cell*. As a result, $P(disk|cell) = 0/3 = 0$, which is not desirable. Consider a document containing the word *disk*. By Equation 2.7, the score of the document categorized as *cell* is 0 because one of the terms, $P(disk|cell)$, is 0. In reality, the document may contain many words that appear in documents of category *cell* and so should be categorized as such.

To avoid the problem introduced by words that never appear in documents of a particular category, Laplace smoothing is used in the calculation of P(Word|Category) as shown below:

$$P_1(W|C) = \frac{1 + N_{WC}}{2 + N_C} \tag{2.5}$$

For example, the number of documents of category *laptop* containing the word *screen*

cell	0.5000
laptop	0.5000

Table 2.3: P(Category) values calculated from Table 2.1

is 3 and the number of documents of category *laptop* is 3. Therefore, $P(screen|laptop) = (1+3)/(2+3) = 4/5 = 0.8000$. Table 2.2 shows P(Word|Category) values calculated by Equation 2.5.

The last step in training a Naive Bayes text classifier is to calculate P(Category) for all categories. Given a category C,

$$P(C) = \frac{N_C}{N} \tag{2.6}$$

- N_C is the number of documents of category C.
- N is the total number of documents in the training set.

Using Table 2.1 as an example, the number of documents of category *cell* is 3 (i.e. D1, D2, and D3). The total number of documents in the training set is 6 (i.e. D1, D2, D3, D4, D5, and D6). Therefore, $P(cell)$ is $3/6 = 0.5000$. Table 2.3 shows P(Category) values calculated from Table 2.1.

2.2.2 Applying

As mentioned before, determining a document's category requires the calculation of scores for all categories. The category with the highest score is the document's category. Having learned the probabilities from the training set, we are now ready to calculate scores for all categories. Below is the formula for calculating the score of document d belonging to category C:

$$score(C) = P(C) \times \prod_{i=1}^{n} (B_{di} P(W_i|C) + (1 - B_{di})(1 - P(W_i|C))) \tag{2.7}$$

Test Doc	Category	*screen*	*coverage*	*disk*	*processor*	*wi-fi*	*signal*
D7	*?*	1	1	0	1	1	1

Table 2.4: Internal representation of a test document

where B_{di} is 1 when word W_i is present in document d and 0 otherwise. Consider the test document in Table 2.4. By Equation 2.7,

$$\begin{aligned}
score(cell) &= P(cell) \times P(screen|cell) \times P(coverage|cell) \times (1 - P(disk|cell)) \\
&\quad \times P(processor|cell) \times P(wi\text{–}fi|cell) \times P(signal|cell) \\
&= 0.5000 \times 0.4000 \times 0.8000 \times (1 - 0.2000) \times 0.4000 \times 0.8000 \times 0.8000 \\
&= 0.5000 \times 0.4000 \times 0.8000 \times 0.8000 \times 0.4000 \times 0.8000 \times 0.8000 \\
&= 0.032768 \\
score(laptop) &= P(laptop) \times P(screen|laptop) \times P(coverage|laptop) \\
&\quad \times (1 - P(disk|laptop)) \times P(processor|laptop) \times P(wi\text{–}fi|laptop) \\
&\quad \times P(signal|laptop) \\
&= 0.5000 \times 0.8000 \times 0.2000 \times (1 - 0.8000) \times 0.8000 \times 0.8000 \times 0.2000 \\
&= 0.5000 \times 0.8000 \times 0.2000 \times 0.2000 \times 0.8000 \times 0.8000 \times 0.2000 \\
&= 0.002048
\end{aligned}$$

Because $score(cell)$ is greater than $score(laptop)$, the test document is categorized as *cell*.

2.3 Multinomial Model

2.3.1 Training

The first step in training a Naive Bayes text classifier is to extract the words from the training set. Table 2.5 shows the internal representation of an example training set. The first column shows the set of document identifiers. The second column shows the document categories. The rest of the columns are the frequencies of the words appearing in the documents. For example, document D1 has 1 occurrence of the word *screen* while document D2 has 1 occurrence of the word *coverage*.

The next step is to calculate P(Word|Category) for all combinations of words and

Training Set	Category	*screen*	*coverage*	*disk*	*processor*	*wi-fi*	*signal*
D1	*cell*	1	2	0	1	1	1
D2	*cell*	0	1	0	0	1	3
D3	*cell*	0	1	0	0	1	2
D4	*laptop*	2	0	1	1	1	0
D5	*laptop*	3	0	1	2	1	0
D6	*laptop*	3	0	2	1	1	0

Table 2.5: Internal representation of the training set

	screen	*coverage*	*disk*	*processor*	*wi-fi*	*signal*
cell	0.0952	0.2381	0.0476	0.0952	0.1905	0.3333
laptop	0.3600	0.0400	0.2000	0.2000	0.1600	0.0400

Table 2.6: P(Word|Category) values calculated by Laplace smoothing

categories. Given a word W and a category C,

$$P_2(W|C) = \frac{N_{WC}}{N_C} \tag{2.8}$$

- N_{WC} is the number of occurrences of word W in documents of category C.
- N_C is the word count of documents of category C.

Using Table 2.5 as an example, the number of occurrences of word *screen* in documents of category *laptop* is $2 + 3 + 3 = 8$. The word count of documents of category *laptop* is 19, obtained by summing the numbers in the last three rows. Therefore, $P(screen|laptop)$ is $8/19 \approx 0.4211$. As another example, the number of occurrences of word *coverage* in documents of category *cell* is $2 + 1 + 1 = 4$. The word count of documents of category *cell* is 15, obtained by summing the numbers in the first three rows. Therefore, $P(coverage|cell)$ is $4/15 \approx 0.2667$.

As with the multi-variate Bernoulli model, Laplace smoothing is used in the calculation of P(Word|Category) to avoid the problem introduced by zero word frequency. The formula for calculating P(Word|Category) is shown below:

$$P_2(W|C) = \frac{1 + N_{WC}}{|V| + N_C} \tag{2.9}$$

where $|V|$ is the number of distinct words in the training set. For example, the number

Test Doc	Category	*screen*	*coverage*	*disk*	*processor*	*wi-fi*	*signal*
D7	*?*	1	2	0	1	1	1

Table 2.7: Internal representation of a test document

of occurrences of word *screen* in documents of category *laptop* is 8, the word count of documents of category *laptop* is 19, and the number of distinct words in the training set is 6. Therefore, $P(screen|laptop) = (1+8)/(6+19) = 9/25 = 0.3600$. Table 2.6 shows $P(\text{Word}|\text{Category})$ values calculated by Equation 2.9.

The last step in training a Naive Bayes text classifier is to calculate $P(\text{Category})$. The formula for calculating $P(\text{Category})$ is identical to that of the multi-variate Bernoulli model. Table 2.3 shows $P(\text{Category})$ values calculated from Table 2.5.

2.3.2 Applying

As with the multi-variate Bernoulli model, determining a document's category requires the calculation of scores for all categories. The category with the highest score is the document's category. Having learned the probabilities from the training set, we are now ready to calculate scores for all categories. Below is the formula for calculating the score of document d belonging to category C:

$$score(C) = P(C) \times \prod_{i=1}^{n} P(W_i|C)^{N_{di}} \tag{2.10}$$

where N_{di} is the number of occurrences of word W_i in document d. Consider the test document in Table 2.7. By Equation 2.10,

$$\begin{aligned} score(cell) &= P(cell) \times P(screen|cell)^1 \times P(coverage|cell)^2 \times P(disk|cell)^0 \\ &\quad \times P(processor|cell)^1 \times P(wi\text{–}fi|cell)^1 \times P(signal|cell)^1 \\ &= 0.5000 \times 0.0952^1 \times 0.2381^2 \times 0.0476^0 \times 0.0952^1 \times 0.1905^1 \times 0.3333^1 \\ &= 0.5000 \times 0.0952 \times 0.0567 \times 1.0000 \times 0.0952 \times 0.1905 \times 0.3333 \\ &= 0.000016314 \end{aligned}$$

$$
\begin{aligned}
score(laptop) &= P(laptop) \times P(screen|laptop)^1 \times P(coverage|laptop)^2 \\
&\quad \times P(disk|laptop)^0 \times P(processor|laptop)^1 \times P(wi\text{–}fi|laptop)^1 \\
&\quad \times P(signal|laptop)^1 \\
&= 0.5000 \times 0.3600^1 \times 0.0400^2 \times 0.2000^0 \times 0.2000^1 \times 0.1600^1 \\
&\quad \times 0.0400^1 \\
&= 0.5000 \times 0.3600 \times 0.0016 \times 1.0000 \times 0.2000 \times 0.1600 \times 0.0400 \\
&= 0.000000369
\end{aligned}
$$

Because $score(cell)$ is greater than $score(laptop)$, the test document is categorized as $cell$.

2.4 Underflow Problem

Sometimes a score is too small to be represented. This happens when the number of words involved in the calculation is large. Such a problem is termed underflow problem. To solve this problem, we applied natural logarithm in the calculation of scores. As shown in the following, the application of natural logarithm does not change the verdict of the text classifier.

2.4.1 Multi-Variate Bernoulli Model

Consider the test document in Table 2.4. By applying logarithm,

$$
\begin{aligned}
score(cell) &= ln(P(cell) \times P(screen|cell) \times P(coverage|cell) \times (1 - P(disk|cell)) \\
&\quad \times P(processor|cell) \times P(wi\text{–}fi|cell) \times P(signal|cell)) \\
&= ln(0.5000 \times 0.4000 \times 0.8000 \times (1 - 0.2000) \times 0.4000 \times 0.8000 \times 0.8000) \\
&= ln(0.5000) + ln(0.4000) + ln(0.8000) + ln(0.8000) + ln(0.4000) \\
&\quad + ln(0.8000) + ln(0.8000) \\
&= -0.6931 - 0.9163 - 0.2231 - 0.2231 - 0.9163 - 0.2231 - 0.2231 \\
&= -3.4181
\end{aligned}
$$

$$
\begin{aligned}
score(laptop) &= ln(P(laptop) \times P(screen|laptop) \times P(coverage|laptop) \\
&\quad \times (1 - P(disk|laptop)) \times P(processor|laptop) \times P(wi\text{–}fi|laptop) \\
&\quad \times P(signal|laptop)) \\
&= ln(0.5000 \times 0.8000 \times 0.2000 \times (1 - 0.8000) \times 0.8000 \times 0.8000 \\
&\quad \times 0.2000) \\
&= ln(0.5000) + ln(0.8000) + ln(0.2000) + ln(0.2000) + ln(0.8000) \\
&\quad + ln(0.8000) + ln(0.2000) \\
&= -0.6931 - 0.2231 - 1.6094 - 1.6094 - 0.2231 - 0.2231 - 1.6094 \\
&= -6.1906
\end{aligned}
$$

Because $score(cell)$ is greater than $score(laptop)$, the test document is categorized as *cell*.

2.4.2 Multinomial Model

Consider the test document in Table 2.7. By applying logarithm,

$$
\begin{aligned}
score(cell) &= ln(P(cell) \times P(screen|cell)^1 \times P(coverage|cell)^2 \times P(disk|cell)^0 \\
&\quad \times P(processor|cell)^1 \times P(wi\text{–}fi|cell)^1 \times P(signal|cell)^1) \\
&= ln(0.5000 \times 0.0952^1 \times 0.2381^2 \times 0.0476^0 \times 0.0952^1 \times 0.1905^1 \\
&\quad \times 0.3333^1) \\
&= ln(0.5000) + 1 \times ln(0.0952) + 2 \times ln(0.2381) + 0 \times ln(0.0476) \\
&\quad + 1 \times ln(0.0952) + 1 \times ln(0.1905) + 1 \times ln(0.3333) \\
&= -0.6931 - 2.3518 - 2.8701 - 0 - 2.3518 - 1.6581 - 1.0987 \\
&= -11.0236
\end{aligned}
$$

$$
\begin{aligned}
score(laptop) &= ln(P(laptop) \times P(screen|laptop)^1 \times P(coverage|laptop)^2 \\
&\quad \times P(disk|laptop)^0 \times P(processor|laptop)^1 \times P(wi\text{–}fi|laptop)^1 \\
&\quad \times P(signal|laptop)^1) \\
&= ln(0.5000 \times 0.3600^1 \times 0.0400^2 \times 0.2000^0 \times 0.2000^1 \times 0.1600^1 \\
&\quad \times 0.0400^1) \\
&= ln(0.5000) + 1 \times ln(0.3600) + 2 \times ln(0.0400) + 0 \times ln(0.2000) \\
&\quad + 1 \times ln(0.2000) + 1 \times ln(0.1600) + 1 \times ln(0.0400) \\
&= -0.6931 - 1.0217 - 6.4378 - 0 - 1.6094 - 1.8326 - 3.2189 \\
&= -14.8135
\end{aligned}
$$

Again, because *score*(*cell*) is greater than *score*(*laptop*), the test document is categorized as *cell*.

2.5 Feature Selection

From the previous example, it is obvious that the speed of categorizing a document is mostly determined by the number of distinct words (i.e. the vocabulary). If the vocabulary size is large, then it would take a lot of time to categorize a document. In order to speed up the categorization, we can reduce the vocabulary involved in the calculation by considering only those words that have highest impact on determining the category of a document. Reducing the size of vocabulary in order to improve performance is termed feature selection.

Let X_i be a random variable for a word in the vocabulary and C be a random variable for all categories. Feature selection is accomplished by selecting the words in the vocabulary that have the highest mutual information between X_i and C. The following is the formula used to calculate the mutual information:

$$
MI(X_i; C) = \sum_{X_i = x_i, C = c} P(x_i, c) ln \frac{P(x_i, c)}{P(x_i)P(c)} \tag{2.11}
$$

where the calculation of the probabilities is dependent on the event model used [2]. For the multi-variate Bernoulli model,

- $P_1(c)$ is the number of documents that belongs to category c divided by the total number of documents,
- $P_1(x_i)$ is the number of documents that contains word x_i divided by the total number of documents, and
- $P_1(x_i, c)$ is the number of documents that belongs to category c and contains word x_i divided by the total number of documents.

For the multinomial model,

- $P_2(c)$ is the word count of documents of category c divided by the total word count,
- $P_2(x_i)$ is the number of occurrences of word x_i divided by the total number of word occurrences, and
- $P_2(x_i, c)$ is the number of occurrences of word x_i in documents of category c divided by the total number of word occurrences.

Chapter 3

Iris: A Naive Bayes Text Classifier

Iris is a Naive Bayes text classifier that was implemented for proof of concept. Although there are many different Naive Bayes text classifiers available for download, most of them are not suitable for our experiments. For example, most of the Naive Bayes text classifiers that we found fall into one of two categories: a command or a software library that the software developers can use to produce a command. The commands, while easier to use than the software libraries, tend to have strict requirements on the structure of the inputs. Also, many of the commands that we found apply only to Email spam classification. Moreover, some commands only work on small inputs; they are not intended for large inputs such as the ones used in our experiments. Software libraries, on the other hand, are a lot harder to use than commands and require steep learning curve. Because of the difficulties associated with the existing Naive Bayes text classifiers, we felt that there is a need to implement one for our purposes.

Iris was implemented in Python. It uses MySQL database to store the statistics of the training and test sets. The advantages of Iris are threefold. First, because Iris was implemented in Python, the source code is executable and therefore does not need to be compiled. Second, because the statistics are stored in the database, the correctness of Iris is easy to verify. Finally, the inputs to Iris have straightforward structure, as evidenced by the following sections.

3.1 Train Iris

Training Iris involves a training set and a set of commands.

```
train/
     cell/
          D1.txt
          D2.txt
          D3.txt
     laptop/
          D4.txt
          D5.txt
          D6.txt
```

Figure 3.1: Example directory structure for training set

3.1.1 Training Set

The training set for Iris is organized into a directory tree. The root directory contains a number of sub-directories, one for each category. The names of the sub-directories are the names of the categories. Each sub-directory in turn contains a set of documents fall under that category. The documents are formatted as plain text.

Figure 3.1 shows the directory structure for an example training set that was used in Chapter 2. As shown in the figure, the root directory `train` contains two sub-directories, one for the `cell` category and the other for the `laptop` category. The `cell` category has 3 documents: `D1.txt`, `D2.txt`, and `D3.txt`. The `laptop` category has 3 documents: `D4.txt`, `D5.txt`, and `D6.txt`. Figure 3.2 shows the contents of the documents.

3.1.2 Commands

Training Iris is realized by three commands, which should be used in the order listed as each command depends on the outputs of the previous commands.

frequency.py

- **Synopsis**:
  `frequency.py` *db user password train_dir*
  - *db* is the database to use, authenticated by *user* and *password*.
  - *train_dir* is the path to a directory that contains the training set.

train/cell/D1.txt

```
screen
coverage coverage
processor
wi-fi
signal
```

train/cell/D2.txt

```
coverage
wi-fi
signal signal signal
```

train/cell/D3.txt

```
coverage
wi-fi
signal signal
```

train/laptop/D4.txt

```
screen screen
disk
processor
wi-fi
```

train/laptop/D5.txt

```
screen screen screen
disk
processor processor
wi-fi
```

train/laptop/D6.txt

```
screen screen screen
disk disk
processor
wi-fi
```

Figure 3.2: Example training set

- **Description**:
  For each document in the training set, `frequency.py` performs three tasks in sequence. First, the words are extracted from the document and the document category is extracted from the document's file path. Second, the frequencies of the words are calculated. Finally, the words, the word frequencies, and the document category are saved into the database.

mutual_info.py

- **Synopsis**:
  `mutual_info.py` *db user password model*
  - *db* is the database to use, authenticated by *user* and *password*.
  - *model* is the event model to use.
- **Description**:
  The mutual information of the words in the database is calculated by using Equation 2.11 and saved into the database. As discussed in Section 2.5, the mutual information is calculated differently depending on the event model used.

```
test/
     cell/
          Dt.txt
```

Figure 3.3: Example directory structure for test set

probability.py

- **Synopsis**:
  `probability.py` *db user password model*
  - *db* is the database to use, authenticated by *user* and *password*.
  - *model* is the event model to use.
- **Description**:
  For each category in the database, P(Category) is calculated and saved into the database. For each combination of the words and categories in the database, P(Word|Category) is also calculated and saved into the database. As discussed in Sections 2.2 and 2.3, P(Word|Category) is calculated differently depending on the event model used. Equation 2.5 is used if the event model is multi-variate Bernoulli and Equation 2.9 is used otherwise.

3.2 Apply Iris

3.2.1 Test Set

The purpose of the test set is to evaluate Iris's accuracy in text classification. The test set consists of documents that are already categorized by experts. For each document, Iris determines the document's category by applying Naive Bayes text classification scheme. For clarity, we hereafter refer to the category determined by Iris as actual category and the category determined by experts as expected category. A document is correctly categorized when the actual category matches the expected category. On the contrary, a document is wrongly categorized when the actual category differs from the expected category. The accuracy is calculated by dividing the number of correctly categorized documents by the total number of documents.

As with the training set, the test set for Iris is organized into a directory tree. The root directory contains a number of sub-directories, one for each category. The

test/cell/Dt.txt

```
screen
coverage coverage
processor
wi-fi
signal
```

Figure 3.4: Example test set

names of the sub-directories are the names of the categories. Each sub-directory in turn contains a set of documents fall under that category. The documents are formatted as plain text.

Figure 3.3 shows the directory structure for an example test set that was used in Chapter 2. As shown in the figure, the root directory `test` contains one sub-directory for the `cell` category. The `cell` category has one document named `Dt`. Figure 3.4 shows the contents of the `Dt` document.

3.2.2 Commands

categorize.py

- **Synopsis**:
  `categorize.py` *db user password model test_dir num_features*
  - *db* is the database to use, authenticated by *user* and *password*.
  - *model* is the event model to use.
  - *test_dir* is the path to a directory that contains the test set.
  - *num_features* is the number of distinct words involved in the categorization.
- **Description**:
  For each document in the test set, the document's category is determined by using the specified event model. The results are displayed in a summary that contains the number of features used, the accuracy of the categorization, and the time it takes to complete the categorization.

```
1   <experiment>
2       <num_features>500</num_features>
3       <accuracy>1.000</accuracy>
4       <execution_time>0.118</execution_time>
5   </experiment>
```

Figure 3.5: Example summary

- **Example**:
  Figure 3.5 shows the summary of running Iris against the test set shown in Figure 3.4. Line 2 shows that `categorize.py` is run with *num_features* set to 500. Line 3 shows that accuracy is 1.000, that is, all documents in the test set are correctly categorized. Finally, line 4 shows that the time it takes to complete the categorization is 0.118 seconds. This is reasonable as there is only one document in the test set.

  Note that the summary is formatted in eXtensible Markup Language (XML) [3]. XML is needed because it facilitates automatic extraction of experiment results. For our purpose, we are interested in 1. the relationship between number of features and accuracy and 2. the relationship between number of features and execution time. In order to observe these relationships, the same experiment must be ran multiple times, each time with a different number of features. The output of the experiments can then be used to generate plots, which can be accomplished by copying the number of features, the accuracy values, and the execution time into a spreadsheet for analysis. In this case, it would be tedious to manually extract the data for each run. By formatting the output as XML, a script can be developed that automatically extracts the data and generates plots. With this approach, not only does the time it takes to analyze the output reduced, but the errors that can potentially arise from manual work are also eliminated. The following describes the script that we developed for automatic generation of plots.

plot.py

- **Synopsis**:
  `plot.py` *multivariate_log multinomial_log*

  - *multivariate_log* is the path to a file that contains the concatenated summaries of categorization by using the multi-variate Bernoulli event model.
  - *multinomial_log* is the path to a file that contains the concatenated summaries of categorization by using the multinomial event model.

- **Description**:
  For each log, the number of features, the accuracy values, and the execution time are extracted. The results are then used to generate two data files. The first data file contains three tab-separated columns: the number of features, the accuracy values extracted from *multivariate_log*, and the accuracy values extracted from *multinomial_log*. The second data file also contains three tab-separated columns: the number of features, the execution time extracted from *multivariate_log*, and the execution time extracted from *multinomial_log*. In order to generate plots, two gnuplot [4] scripts are also generated, one for each data file. The gnuplot scripts are executed to produce the plots.

3.3 Search Documents by GUI

Besides performance evaluation, Iris also facilitates document searching through a GUI. The following subsections describe the command used to bring up the GUI and the approach used to search and view documents of interest.

3.3.1 Command

gui.py

- **Synopsis**:
  `gui.py` *db user password model num_features*
  - *db* is the database to use, authenticated by *user* and *password*.
  - *model* is the event model to use.
  - *num_features* is the number of distinct words involved in the categorization.

- **Description**:
  Executing the command will bring up the GUI as shown in Figure 3.6. The GUI consists of a `File` menu and three panels. The `File` menu is used to open

Figure 3.6: Iris GUI startup

a directory that contains a set of documents that the user wishes to categorize. Once the documents are categorized, the user can then use the controls within the three panels for searching and viewing documents of interest. For example, the left panel displays a set of document categories for the user to choose from. The right panel contains a query field, a `Query` button, and a blank area for displaying the search results whenever the `Query` button is clicked. The bottom panel contains a blank area for displaying a document in its entirety.

3.3.2 Approach

For demontration purposes, the test set from the Reuters-21578 dataset [5] is used. The following describes the step-by-step procedure of initiating the document searching process. First, the GUI is brought up by using the command syntax described in the previous subsection. Second, the test set is categorized by clicking on the `Open Directory...` menu item located under the `File` menu. This prompts the user for the directory that contains a set of documents that he or she wishes to categorize. In this case, the directory that contains the test set is selected. Third, a progress dialog is displayed as shown in Figure 3.7. The purpose of this dialog is to inform the user

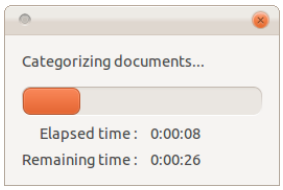


Figure 3.7: A progress bar for text categorization

the progress of text categorization in terms of the elapsed time and the remaining time. Finally, the user can begin the search process by either selecting a category from the left panel or entering a query in the query field. There are three approaches for searching documents:

- *searching by a query string*: returns all documents that contain the query string, regardless of the document categories. With this approach, the user enters the query in the query field and checks all categories in the left panel. Figure 3.8 shows an example of searching by a query string.

- *searching by categories*: returns all documents that are of the specified categories. With this approach, the user leaves the query field blank and checks the categories of interest in the left panel. Figure 3.9 shows an example of searching by categories.

- *searching by a query string and categories*: returns all documents that contain the query string and are of the specified categories. With this approach, the user enters the query in the query field and checks the categories of interest in the left panel. Figure 3.10 shows an example of searching by a query string and categories.

For each matched document, the right panel displays three items: the path to the document, the document category enclosed in parentheses, and the first 500 characters of the document. The user can click on the path to view a document in its entirety by clicking on a document path. Figure 3.11 shows a full document displayed in the bottom panel.

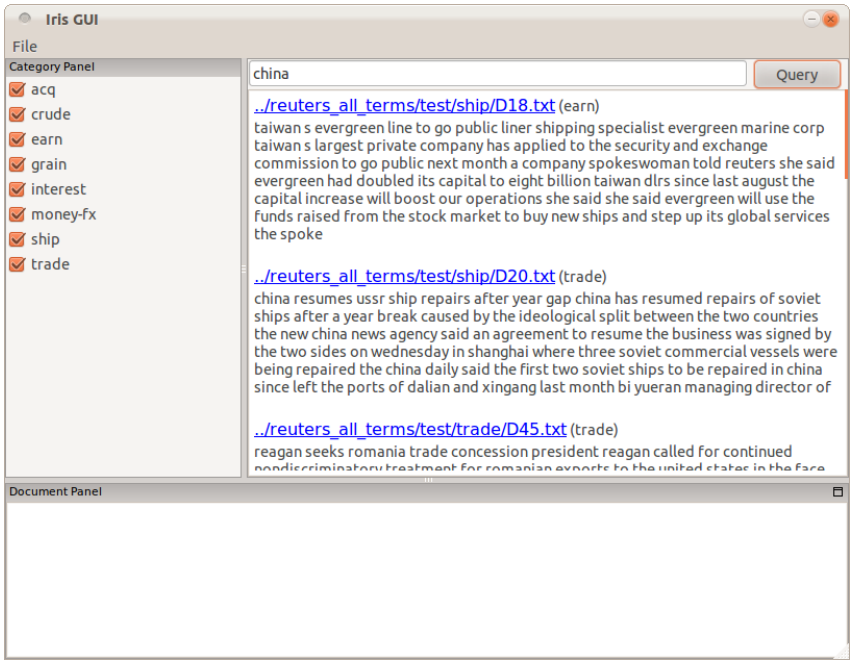


Figure 3.8: Screenshot for searching by query string

Figure 3.9: Screenshot for searching by category

Figure 3.10: Screenshot for searching by query string and category

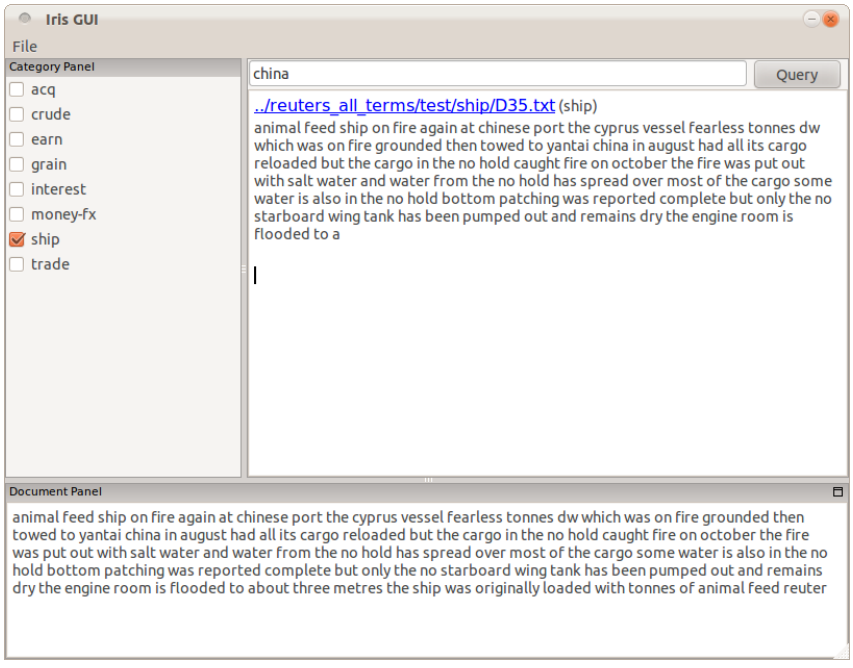


Figure 3.11: Screenshot for viewing document

Chapter 4

Iris Design and Implementation

Iris consists of four components: a database, a `util` module, five commands, and a Graphical User Interface (GUI). The database is used to store the statistics of the training set, the `util` module contains a set of helper functions that are called by the commands, the commands are executed by the user to train and test Iris, and the GUI facilitates document searching through keywords and categories.

4.1 The Database

Figure 4.1 shows the Entity/Relationship diagram, hereafter referred to as E/R diagram, of the Iris database. Three different shapes are used to represent the three different kinds of elements within the E/R diagram: rectangles represent entities, diamonds represent relationships, and ovals represent attributes. The attributes with an underline are primary keys, which are unique identifiers for database records. Although the figure distinguishes entities from relationships, they are all referred to as tables in database terminology. The script used to create the database tables can be found in Appendix A.

4.1.1 The `word` Table

This table stores all distinct words in the training set. Each `word` record has three attributes: `id`, `word` and `mutual_info`. `id` is an unique identifier for the word. `word` contains the string representation of the word. `mutual_info` contains the word's mutual information.

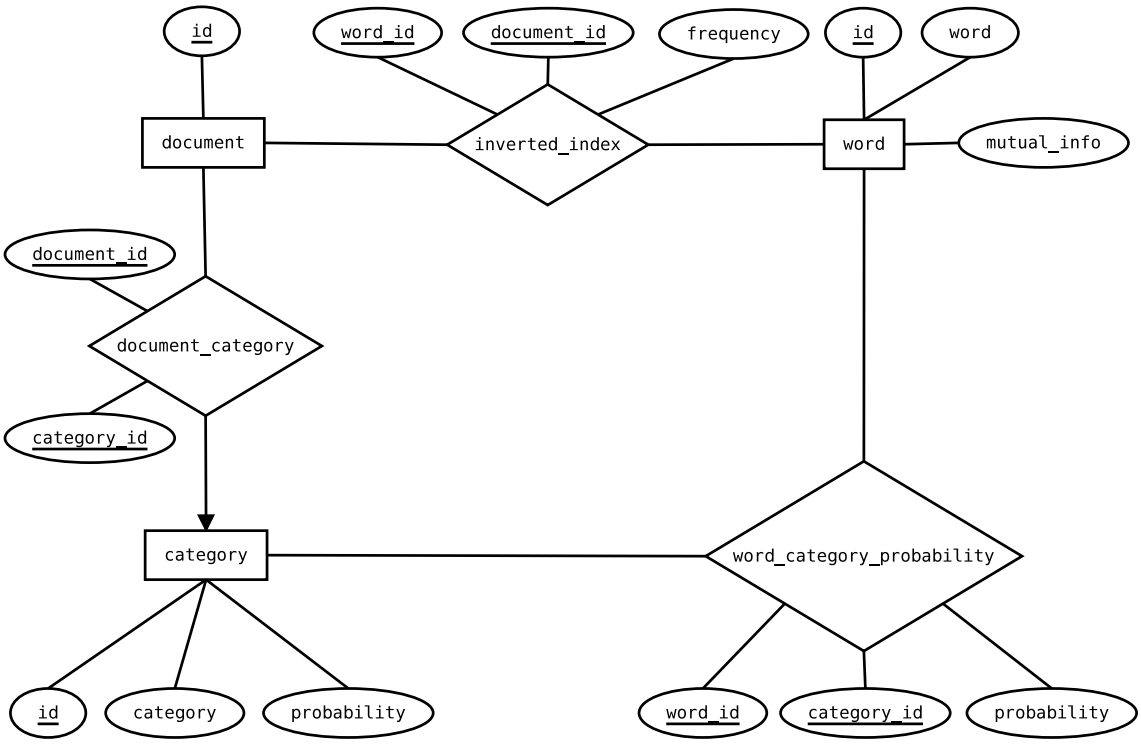


Figure 4.1: E/R diagram: rectangles represent entities, diamonds represent relationships, and ovals represent attributes. Arrow means one-to-many, otherwise many-to-many.

4.1.2 The `document` Table

This table stores all documents in the training set. Each `document` record has only one attribute `id`, which is used as the unique identifier for a document.

4.1.3 The `category` Table

This table stores all categories in the training set. Each `category` record has three attributes: `id`, `category`, and `probability`. `id` is an unique identifier for the category. `category` contains the string representation of the category. `probability` contains $P(\text{Category})$ of the category.

4.1.4 The `document_category` Table

This table is responsible for associating each document with its category. Each `document_category` record has two attributes: `document_id` refers to the `id` attribute of the `document` table. `category_id` refers to the `id` attribute of the `category` table. Note that in Figure 4.1, the line linking `document_category` and `category` has an arrow head whereas the line linking `document_category` and `document` does not. This means that the relation between `category` and `document` is one-to-many, that is, each category can have many documents but each document belongs only to one category.

4.1.5 The `inverted_index` Table

This table stores the frequencies of the words for all documents in the training set. Each `inverted_index` record has three attributes: `word_id`, `document_id`, and `frequency`. `word_id` refers to the `id` attribute of the `word` table. `document_id` refers to the `id` attribute of the `document` table. `frequency` is the frequency of the word of ID `word_id` in the document of ID `document_id`. Note that in Figure 4.1, neither of the line linking `inverted_index` and `document` and the line linking `inverted_index` and `word` has an arrow head. This means that the relation between `document` and `word` is many-to-many, that is, each document can have many different words and each distinct word can appear in many documents.

4.1.6 The `word_category_probability` Table

This table stores P(Word|Category) for all combinations of words and categories in the training set. Each `word_category_probability` record has three attributes: `word_id`, `category_id`, and `probability`. `word_id` refers to the `id` attribute of the `word` table. `category_id` refers to the `id` attribute of the `category` table. `probability` is P(Word|Category) for the word of ID `word_id` and the category of ID `category_id`. Note that in Figure 4.1, neither of the line linking `word_category_probability` and `word` and the line linking `word_category_probability` and `category` has an arrow head. This means that the relation between `word` and `category` is many-to-many, that is, each unique word can appear in documents of different categories and vice versa.

4.2 The `util` Module

The `util` module, as listed in Appendix B.1, consists of a `DB` class for interacting with the database and a `get_word_frequencies` function for reading a document in the training or test set.

4.2.1 The `DB` Class

The `DB` class provides five different services: connecting to the database, disconnecting from the database, inserting data, updating data, and querying data. Each method in the `DB` class, except for the methods for connecting to and disconnecting from the database, uses a Structured Query Language (SQL) statement to communicate with the database. The advantage of using class is information hiding, that is, class members can be created to keep track of certain information that would otherwise be too complex for the caller of the `DB` class to manage. In our design, category IDs and word IDs are managed by the `DB` class. Therefore, the caller of the `DB` class does not need to provide category ID or word ID whenever it inserts data into the `category` or the `word` table.

Connecting to the database is accomplished by the `DB` class constructor which takes three parameters: `database`, `user`, and `password`. `database` is the name of the database to connect, authenticated by `user` and `password`. Disconnecting from the database is accomplished by the `close` method which takes no parameters.

There are six methods for inserting data into the database, one for each database table. Noteworthy are the two methods called `insert_category` and `insert_word`. The `insert_category` method inserts a category into the `category` table. As mentioned before, each `category` record contains three attributes: `id`, `category`, and `probability`. Since the caller may not know the probability of a category when inserting it, the probability is set to 0.0. Similarly, the `insert_word` method inserts a word into the `word` table. As mentioned before, each `word` record contains three attributes: `id`, `word`, and `mutual_info`. Since the caller may not know the mutual information of a word when inserting it, the mutual information is set to 0.0.

Setting the probability of a category and the mutual information of a word are accomplished by methods: `update_category` and `update_word`. The `update_category` method takes two parameters: `cid` and `p`, which are the ID and probability of the category to be updated respectively. The `update_word` method also takes two parameters: `wid` and `m`, which are the ID and mutual information of the word to be updated respectively.

Finally, there are fourteen methods for querying data from the database. Below is a brief description for each of the fourteen methods. The naming conventions used here are that `w` is the string representation of a word, `wid` is the ID of a word, `c` is the string representation of a category, and `cid` is the ID of a category.

- `get_category_id(c)`: returns the ID of category `c` if it exists in the `category` table and 0 otherwise.
- `get_word_id(w)`: returns the ID of word `w` if it exists in the `word` table and 0 otherwise.
- `get_word_dict(num_words)`: returns a dictionary where the keys are words and the values are word IDs. Only the words with top `num_words` mutual information are included in the dictionary.
- `get_category_dict()`: returns a dictionary where the keys are categories and the values are category IDs.
- `get_num_documents_of_word(wid)`: returns the number of documents containing word of ID `wid`.
- `get_num_documents_of_category(cid)`: returns the number of documents belonging to category of ID `cid`.

- `get_num_documents_of_word_category(wid,cid)`: returns the number of documents that belong to a category of ID `cid` and contain a word of ID `wid`.
- `get_num_documents()`: returns the number of documents in the `document` table.
- `get_word_category_frequency(wid,cid)`: returns the number of occurrences of a word of ID `wid` in the documents belonging to a category of ID `cid`.
- `get_category_frequency(cid)`: returns the word count of the documents belonging to a category of ID `cid`.
- `get_word_frequency(wid)`: returns the number of occurrences of a word of ID `wid` in all documents.
- `get_frequency()`: returns the word count of all documents.
- `get_category_probabilities()`: returns a dictionary where the keys are category IDs and the values are P(Category)s. Note that there is no method which returns only one P(Category). The reason is that getting all P(Category)s at once is more efficient than getting P(Category) one at a time.
- `get_word_category_probabilities()`: returns a dictionary of dictionaries where the keys for the first-level dictionary are word IDs and the keys and values for the second-level dictionaries are category IDs and P(Word|Category)s respectively. Note that there is no method which returns only one P(Word|Category). The reason is that getting all P(Word|Category)s at once is more efficient than getting P(Word|Category) one at a time.

4.2.2 The `get_word_frequencies` Function

The purpose of the `get_word_frequencies` function is to return the frequencies of each distinct word in a document. The function takes parameter `document_body`, which is a string containing the content of a document, as input. Words are extracted by splitting `document_body` into a list of tokens and removing the trailing punctuation in the tokens if there are any. A word is considered valid if and only if it consists of nothing but English alphabet, apostrophe, and dash.

4.3 The Commands

There are two sets of commands: one for training Iris and the other for testing Iris. Training Iris requires the use of the `DB` class to insert the statistics of the training set into the database. Testing Iris, on the other hand, requires the use of the `DB` class to query the statistics of the training set from the database.

4.3.1 frequency.py

For each document in the training set, `frequency.py`

1. creates an ID for the document and inserts it into the `document` table.
2. extracts the document category from the document's file path and inserts it into the `category` table by calling the `insert_category` function.
3. relates the document with the document category by inserting their IDs into the `document_category` table.
4. extracts the words from the document and inserts each unique word into the `word` table by calling the `insert_word` function.
5. calculates the word frequencies for each unique word and inserts them into the `inverted_index` table.

`frequency.py` is listed in Appendix B.2.

4.3.2 mutual_info.py

For each word in the `word` table, the word's mutual information is calculated and inserted into the `mutual_info` attribute of the `word` table by calling the `update_word` function. As mentioned before, the mutual information is calculated as follows:

$$MI(X_i; C) = \sum_{X_i=x_i, C=c} P(x_i, c) log \frac{P(x_i, c)}{P(x_i)P(c)}$$

where X_i is a random variable for a word in the `word` table, C is a random variable for all categories, and the calculation of the probabilities is dependent on the event model used. For the multi-variate Bernoulli model,

- $P(c)$: `get_num_documents_of_category(`c`)` / `get_num_documents()`.
- $P(x_i)$: `get_num_documents_of_word(`x_i`)` / `get_num_documents()`.
- $P(x_i, c)$: `get_num_documents_of_word_category(`x_i`,`c`)` / `get_num_documents()`.

For the multinomial model,

- $P(c)$: `get_category_frequency(`c`)` / `get_frequency()`.
- $P(x_i)$: `get_word_frequency(`x_i`)` / `get_frequency()`.
- $P(x_i, c)$: `get_word_category_frequency(`x_i`,`c`)` / `get_frequency()`.

`mutual_info.py` is listed in Appendix B.3.

4.3.3 probability.py

`probability.py` calculates P(Category) and P(Word|Category) and saves them into the database so that they can be used to derive *score*(Category).

For each category C in the `category` table, $P(C)$ is calculated and inserted into the `probability` attribute of the `category` table by calling the `update_category` function. $P(C)$ is calculated by dividing `get_num_documents_of_category(`C`)` by `get_num_documents()`. Logarithm is applied to $P(C)$ to prevent underflow error.

For each word W and category C in all combinations of the words in the `word` table and categories in the `category` table, P(W|C) is calculated and inserted into the `probability` attribute of the `word_category_probability` table. As with $P(C)$, logarithm is applied to P(W|C) to prevent underflow error. Again, the calculation of P(W|C) is dependent on the event model used. Let `wid` be the ID of word W, `cid` be the ID of category C, and `V` be the number of rows in the `word` table. For the multi-variate Bernoulli model,

1. 1 + `get_num_documents_of_word_category(wid,cid)`.
2. 2 + `get_num_documents_of_category(cid)`.
3. P(W|C) is the result of step 1 divided by the result of step 2.

For the multinomial model,

1. 1 + `get_word_category_frequency(wid,cid)`.

2. `V` + `get_category_frequency(cid)`.

3. P(W|C) is the result of step 1 divided by the result of step 2.

`probability.py` is listed in Appendix B.4.

4.3.4 categorize.py

`categorize.py` determines the categories of the documents in the test set by using the statistics of the training set. For performance reasons, it uses only N words in the `word` table where N is specified by the user. The N words are obtained by calling the `get_word_dict` function with the function parameter `num_words` set to N.

For each document in the test set, determining the category of the document requires the use of P(Category) and P(Word|Category). P(Category) is obtained by calling the `get_category_probabilities` function. P(Word|Category) is obtained by calling the `get_word_category_probabilities` function. `categorize.py` is listed in Appendix B.5.

Text categorization can be a time consuming process when the test set is large. This is so because for each document in the test set, `categorize.py` has to compute the score of the document belonging to a category for all categories in the `category` table and labels the document with the category of the highest score. With such a computationally intensive task, the usual approach for improving the performance is through the use of multiprocessing. This could potentially drastically reduce the time for text categorization, depending on the number of cores contained within the Central Processing Unit (CPU) of the computer used. Here we used a Python module called `Pool` that creates a fixed number of processes, each responsible for processing a portion of the test set. With this approach, the ideal performance could be achieved when each process runs in a separate CPU core.

4.4 The GUI

Figure 4.2 shows the call graph of `gui.py`, the Python executable that is used to start the GUI. The italicized text refers to the possible operations that can be performed by the user. In total there are five operations: *startup*, *open*, *quit*, *query*, and *view document*. Below each operation is a list of functions that are invoked. Tabbing is used to show the direction of function invocation. For example, lines 3 and 4 show

that `IrisGUI.__init__()` is invoked by `IrisApp.__init__()`. Similarly, lines 4 and 5 show that `DB.get_category_dict()` is invoked by `IrisGUI.__init__()`.

The tasks carried out by each operation is described as follows:

- The *startup* operation (lines 1 through 6): starts up the GUI. This involves creating a `DB` object for connecting to the database, populating the window with various menu items and panels, extracts all the categories from the database, and lists the categories in the left panel.
- The *open* operation (lines 7 through 14): opens the directory that contains the test set. It triggers `IrisGUI.on_open()` which invokes the necessary functions for categorizing the test set.
- The *quit* operation (lines 15 through 17): quits the GUI. It triggers `IrisGUI.on_quit()` which invokes `DB.close()` for closing the database connection.
- The *query* operation (lines 18 through 21): queries the test set with specified keywords and categories and displays the result in the right panel. It triggers `IrisGUI.on_query()` which invokes `is_match()` for finding all matched documents.
- The *view document* operation (lines 22 and 23): displays a document in the bottom panel. It triggers `IrisGUI.on_click()`.

```
1   startup
2       DB.__init__()
3       IrisApp.__init__()
4           IrisGUI.__init__()
5               DB.get_category_dict()
6       IrisApp.MainLoop()
7   open
8       IrisGUI.on_open()
9           get_document_paths()
10          DB.get_category_probabilities()
11          DB.get_word_category_probabilities()
12          DB.get_category_dict()
13          DB.get_word_dict()
14          categorize()
15  quit
16      IrisGUI.on_quit()
17          DB.close()
18  query
19      IrisGUI.on_query()
20          DB.get_category_dict()
21          is_match()
22  view document
23      IrisGUI.on_click()
```

Figure 4.2: Call graph of `gui.py`

Chapter 5

Performance Evaluation

As mentioned before, Naive Bayes has two variants depending on the event model used: multi-variate Bernoulli and multinomial. Here we are interested in their performance given datasets of various characteristics. From the performance results we can then determine which event model is more suitable for text classification in general. With this goal in mind, the following three metrics are used for performance evaluation: 1. how long does it take to train the text classifier, 2. how accurate is the text classifier, and 3. how long does it take for the text classifier to categorize a test set. To find out how these two event models score on these three metrics, three popular datasets from the text classification literature are used. The following describes how each of these datasets are collected and how they are organized.

5.1 Datasets

5.1.1 20 Newsgroups

The 20 Newsgroups dataset [6] is a set of newsgroup documents extracted from 20 different newsgroups. Reportedly collected by Ken Lang, the dataset contains approximately 20,000 documents which are nearly equally divided into the 20 newsgroups. The newsgroups serve as document categories and each newsgroup is identified by the topic under discussion. In total there are six groups of closely related topics. For example, one of the groups is computers. This group contains five closely related topics such as computer graphics, the Microsoft Windows operating system, the IBM PC hardware, the Macintosh hardware, and the X window system. A complete listing of the topics and their groupings are shown in Table 5.1.

comp.graphics comp.os.ms-windows.misc comp.sys.ibm.pc.hardware comp.sys.mac.hardware comp.windows.x	rec.autos rec.motorcycles rec.sport.baseball rec.sport.hockey	sci.crypt sci.electronics sci.med sci.space
misc.forsale	talk.politics.misc talk.politics.guns talk.politics.mideast	talk.religion.misc alt.atheism soc.religion.christian

Table 5.1: Topics for 20 Newsgroups

The 20 Newsgroups dataset is made available as a compressed `tar` file. Decompressing and unpacking the file create two directories: one contains the training set and the other the test set. Both directories have identical structures:

- The root directory contains a number of sub-directories, one for each newsgroup.
- The names of the sub-directories are the topics of the newsgroups.
- Each sub-directory in turn contains a set of documents fall under that topic.
- A document name consists of a series of five or six digits.

5.1.2 Reuters-21578

The Reuters-21578 Distribution 1.0 dataset, Reuters-21578 [5] for short, is a set of news stories published from Reuters Ltd. in 1987. Made available to the public in 1990, the original dataset contains $22,173$ news stories and is therefore referred to as Reuters-22173. It served as a benchmark for evaluating various text classification algorithms. However, many news stories in Reuters-22173 are duplicates which can potentially distort the results of text classification. As a result, a new dataset was created by removing the duplicates that was present in Reuters-22173. This reduced the number of news stories to $21,578$ and therefore the new dataset is referred to as Reuters-21578.

The news stories in Reuters-21578 are categorized by people from Reuters Ltd. and Carnegie Group Inc. In total there are five types of categories as listed below:

- `EXCHANGES`: a set of stock exchanges.
- `ORGS`: a set of organizations.

- PEOPLE: a set of people's names.
- PLACES: a set of places.
- TOPICS: a set of economic subjects.

Our experiment uses the TOPICS categories.

Reuters-21578 is made available as a compressed tar file. Decompressing and unpacking the file create a set of files with the following extensions:

- sgm: an Standard Generalized Markup Language (SGML) file which stores the news stories and the associated metadata. The news stories are organized into 1,000 per file. For example, the first 1,000 news stories can be found in reut2-000.sgm, the second 1,000 news stories can be found in reut2-001.sgm, etc. For completeness, the last 578 news stories can be found in reut2-021.sgm.
- dtd: a Document Type Definition (DTD) file which describes the structure of the SGML files. It defines that a news story always starts with <REUTERS> and ends with </REUTERS>. Within <REUTERS> are two attributes called TOPICS and LEWISSPLIT. The possible values for attribute TOPICS are YES, NO, and BYPASS. The possible values for attribute LEWISSPLIT are TRAINING, TEST, and NOT-USED. Together these two attributes specify how the text corpus is split into training and test sets. In total there are three different ways of splitting the text corpus: the modified Lewis (ModLewis) split, the modified Apte (ModApte) split, and the modified Hayes (ModHayes) split. Our experiment uses the ModApte split.

  The children of a REUTERS element are:

  - DATE: specifies the date and time when the news story was published.
  - TOPICS: specifies the economic subject(s) which the news story is categorized into.
  - PLACES: specifies the place(s) which the news story is categorized into.
  - PEOPLE: specifies the people's name(s) which the news story is categorized into.
  - ORGS: specifies the organization(s) which the news story is categorized into.
  - EXCHANGES: specifies the stock exchange(s) which the news story is categorized into.

	student	faculty	staff	department	course	project	other	Total
cornell	128	34	21	1	44	20	619	867
texas	148	46	3	1	38	20	571	827
washington	126	31	10	1	77	21	939	1,205
wisconsin	156	42	12	1	85	25	942	1,263
misc	1,083	971	91	178	686	418	693	4,120
Total	1,641	1,124	137	182	930	504	3,764	8,282

Table 5.2: Categories for WebKB

  - `TEXT`: encloses the news story as is.

  Note that a news story may be indexed with more than one category. As a result, Reuters-21578 is referred to as a multi-labelled dataset.

- `txt`: a text file which lists the categories of a particular type. In total there are five text files, one for each type of categories:

  - `all-exchanges-strings.lc.txt`: lists 39 categories of type `EXCHANGES`.
  - `all-orgs-strings.lc.txt`: lists 56 categories of type `ORGS`.
  - `all-people-strings.lc.txt`: lists 267 categories of type `PEOPLE`.
  - `all-places-strings.lc.txt`: lists 175 categories of type `PLACES`.
  - `all-topics-strings.lc.txt`: lists 135 categories of type `TOPICS`.

5.1.3 WebKB

The 4 Universities dataset, WebKB [7] for short, is a set of web pages extracted from a number of universities. Collected by the CMU text learning group, the dataset contains $8,282$ web pages which are distributed across seven categories. Table 5.2 shows the distribution of the web pages both in terms of the categories and the universities. The horizontal table headers list the categories while the vertical table headers list the universities where the web pages were extracted. For example, column 1, row 1 of the table shows that university `cornell` has 128 web pages of category `student`. Note that university `misc` represents a group of miscellaneous universities that the CMU text learning group did not name explicitly.

WebKB is made available as a compressed `tar` file. Decompressing and unpacking the file create a three-level directory structure:

- The root directory contains seven second-level directories, one for each category.
- The names of the second-level directories are the categories.
- Each second-level directory contains five third-level directories, one for each university.
- The names of the third-level directories are the universities.
- Each third-level directory contains a set of web pages of corresponding category and university.
- A web page is named after its Uniform Resource Locator (URL) with forward slashes replaced by carets.

Note that the WebKB dataset is not split into training and test sets. It is up to the text classification researchers to do the split.

5.2 Data Processing

While the aforementioned datasets make it possible to compare the performance of various text classifiers, they are imperfect in many respects. For example, the Reuters-21578 dataset is difficult to use for three reasons. First, a lot of information presented in the SGML and text files are not needed. Second, it is not easy to extract the categories and the news stories from the SGML files; doing so would require us to construct an SGML parser. Finally, the rules for splitting the text corpus into training and test sets are not always clear. As a result, we used the datasets provided by Cardoso-Cachopo [8] instead.

Cardoso-Cachopo processed the datasets and made them available online so that other text classification researchers can benefit from her work. The most significant improvement Cardoso-Cachopo has made is that all the processed datasets have identical data structures. For example, each processed dataset is stored into two text files: one contains the training set and the other contains the test set. The text file contains one document per line. Each line starts with a category followed by a corresponding document. Because all the data structures are identical, only one script is needed to convert the data structure into the one acceptable by Iris. Other improvements that Cardoso-Cachopo has made are described as follows:

Category	Train	Test	Total
alt.atheism	480	319	799
comp.graphics	584	389	973
comp.os.ms-windows.misc	572	394	966
comp.sys.ibm.pc.hardware	590	392	982
comp.sys.mac.hardware	578	385	963
comp.windows.x	593	392	985
misc.forsale	585	390	975
rec.autos	594	395	989
rec.motorcycles	598	398	996
rec.sport.baseball	597	397	994
rec.sport.hockey	600	399	999
sci.crypt	595	396	991
sci.electronics	591	393	984
sci.med	594	396	990
sci.space	593	394	987
soc.religion.christian	598	398	996
talk.politics.guns	545	364	909
talk.politics.mideast	564	376	940
talk.politics.misc	465	310	775
talk.religion.misc	377	251	628
Total	11,293	7,528	18,821

Table 5.3: Training set versus test set for 20 Newsgroups

Category	Train	Test	Total
acq	1,596	696	2,292
crude	253	121	374
earn	2,840	1,083	3,923
grain	41	10	51
interest	190	81	271
money-fx	206	87	293
ship	108	36	144
trade	251	75	326
Total	5,485	2,189	7,674

Table 5.4: Training set versus test set for Reuters 21578

Category	Train	Test	Total
course	620	310	930
faculty	750	374	1,124
project	336	168	504
student	1,097	544	1,641
Total	2,803	1,396	4,199

Table 5.5: Training set versus test set for WebKB

- 20 Newsgroups: Attachments and duplicates that are found within the news documents were eliminated. Table 5.3 shows the distribution of documents across the training and test sets.

- Reuters-21578: News stories that belong to more than one category were eliminated. Also, two new datasets were created out of the existing dataset due to the large number of categories. The first dataset is called R8 because it contains only the 8 most frequent categories. The second dataset is called R52 because it contains only the 52 most frequent categories. Our experiment uses the R8 dataset. Table 5.4 shows the distribution of documents across the training and test sets.

- WebKB: The `staff` and `department` categories are eliminated because they have significantly fewer web pages than the rest of the categories. Also, the `other` category is also eliminated because its corresponding web pages are very different from each other. Moreover, the dataset is randomly split into training and test sets. The resulting training and test sets contain two thirds and one third of the web pages respectively. Table 5.5 shows the distribution of documents across the training and test sets.

For our experiment, we used the following five datasets: `20news_stemmed`, `reuters_stemmed`, `webkb_stemmed`, `20news_all_terms`, and `reuters_all_terms`. The first three datasets are obtained by eliminating the stop words and those words that contain less than three characters. The last two datasets are obtained by retaining all the words found in the documents.

Dataset	Multi-variate Bernoulli	Multinomial
20news_stemmed	15,190.583	16,538.395
reuters_stemmed	1,302.091	1,111.936
webkb_stemmed	584.850	612.389
20news_all_terms	30,026.035	30,593.160
reuters_all_terms	2,097.127	2,077.630

(a) Total

Dataset	Multi-variate Bernoulli	Multinomial
20news_stemmed	1.345	1.464
reuters_stemmed	0.221	0.189
webkb_stemmed	0.209	0.218
20news_all_terms	2.659	2.709
reuters_all_terms	0.357	0.353

(b) Per document

Figure 5.1: Training time (in seconds)

Dataset	Word count
20news_stemmed	142.649
reuters_stemmed	66.324
webkb_stemmed	133.570
20news_all_terms	269.016
reuters_all_terms	105.279

Figure 5.2: Average word count per document

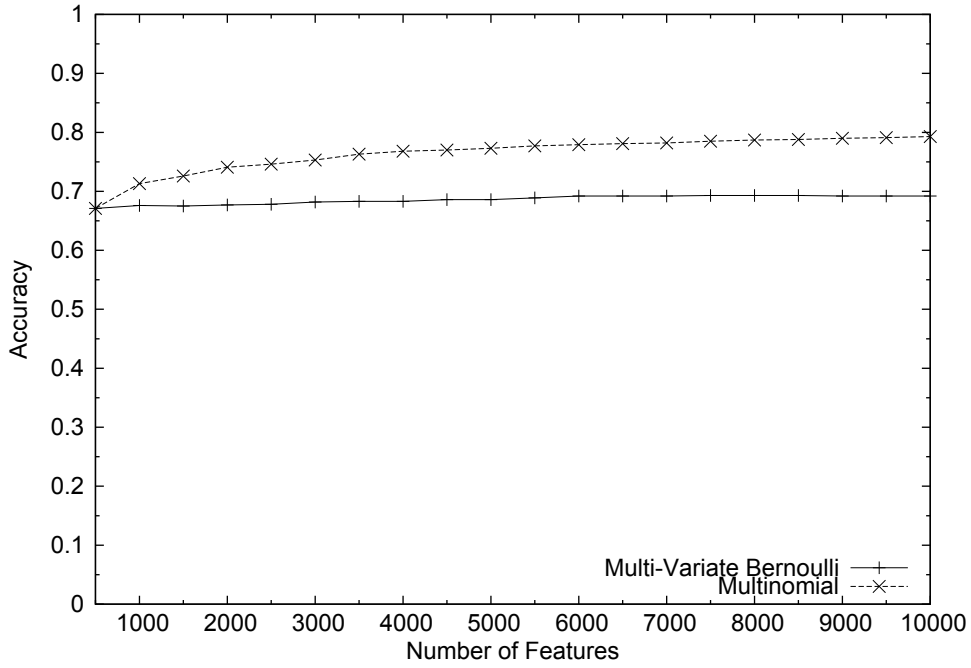
5.3 Training Results

The machine which we used to run our experiment is a HP Pavilion PC with an AMD Athlon 64 X2 Dual Core Processor 5200+ and 2 gigabytes of RAM. Figure 5.1 (a) shows the time it took to train Iris against the datasets. The difference in the training time between multi-variate Bernoulli and multinomial is minimal. On the other hand, the training time varies widely across the five datasets. Training the `20news_all_terms` dataset took the longest amount of time, about 8.341 hours. Training the `webkb_stemmed` dataset took the shortest amount of time, about 0.162 hours. This is so because the `20news` dataset has a lot more documents in the training set than does the `webkb` dataset. Figure 5.1 (b) shows the average training time per document. The `20news_all_terms` dataset still took significant longer to finish than the rest of the datasets. This may be caused by the different word counts per document, as shown in Figure 5.2. For example, the `20news_all_terms` dataset has the largest word count per document compared to the rest of the datasets. As a result, it has the highest average training time per document. On the contrary, it is not clear why the `webkb_stemmed` dataset—with the word count per document twice as that of the `reuters_stemmed` dataset—has similar average training time per document as that of the latter.

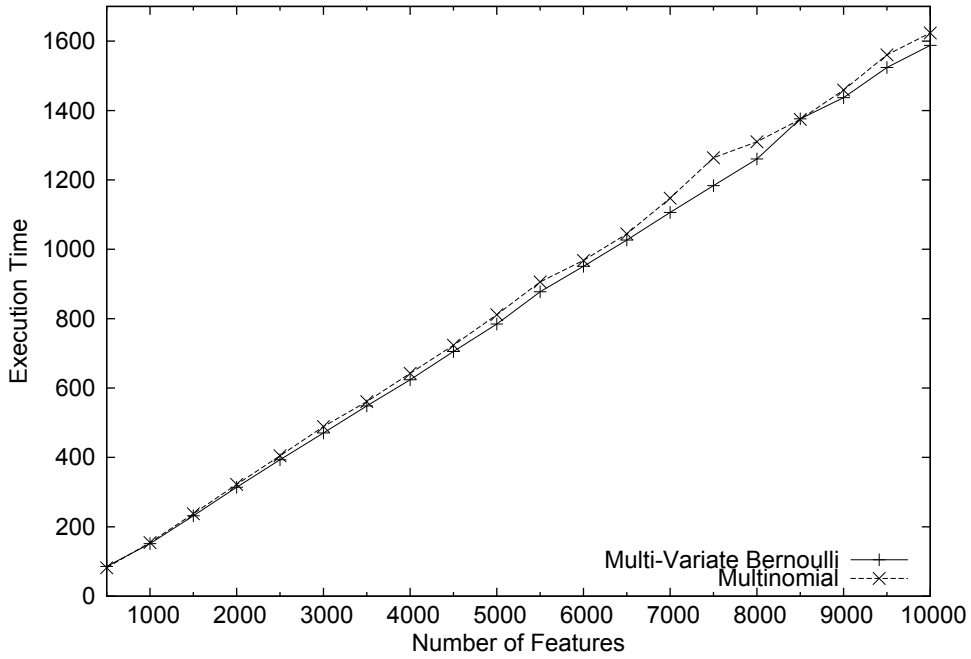
5.4 Test Results for Stemmed

For each dataset, we are interested in the accuracy achieved by the event models and the time it took to categorize the test set, hereafter referred to as the categorization time. We are also interested in seeing how the number of features affect the accuracy and the categorization time. We therefore ran the test multiple times, each time with different number of features. With this approach, the first test was run with the number of features set to 500. The rest of the tests were run with an increment of 500, with the highest number of features attempted being $10,000$. The test results were then used to generate two plots per dataset: one for accuracy and one for categorization time. For example, the `20news_stemmed` dataset has two plots as shown in Figures 5.3 (a) and 5.3 (b). The first plot shows accuracy with respect to number of features. The second plot shows categorization time with respect to number of features. Note that the time is in seconds.

Figures 5.3 (a), 5.4 (a), and 5.5 (a) show the accuracy of the `20news_stemmed`,

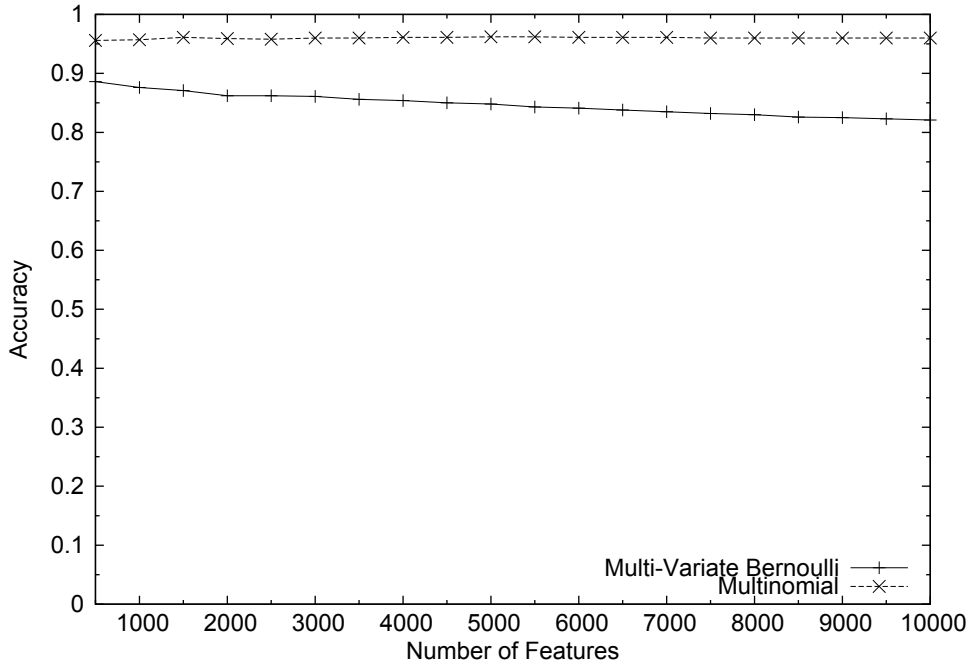


(a) Accuracy

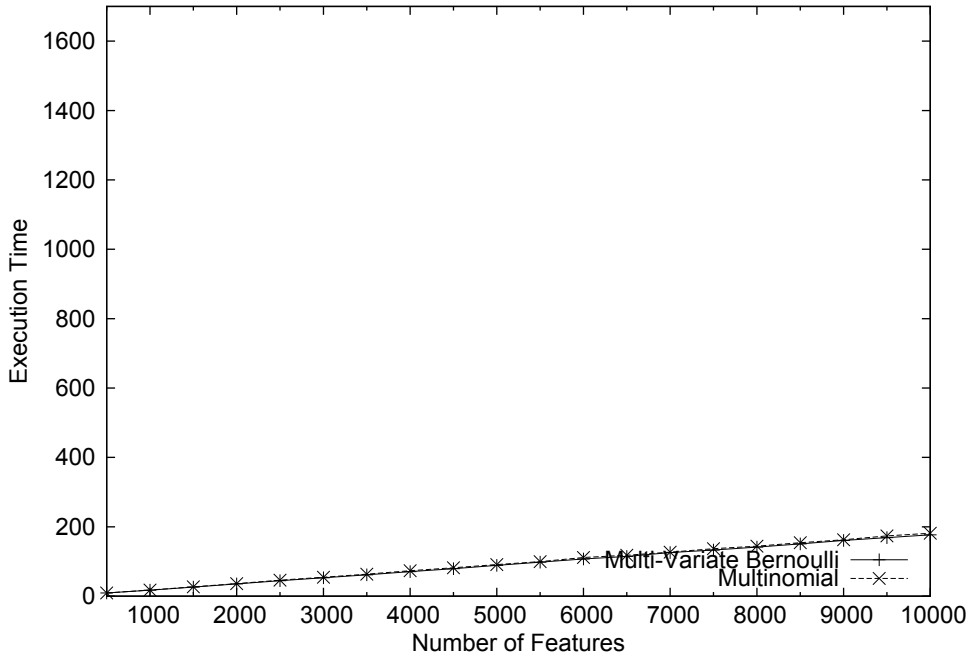


(b) Categorization time (in seconds)

Figure 5.3: Test results for `20news_stemmed`

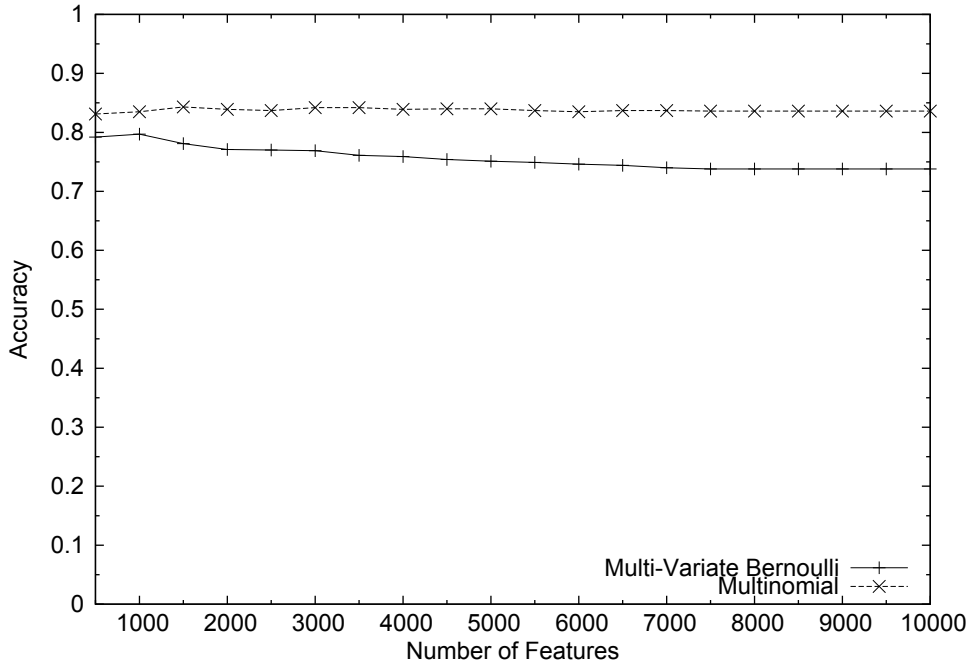


(a) Accuracy

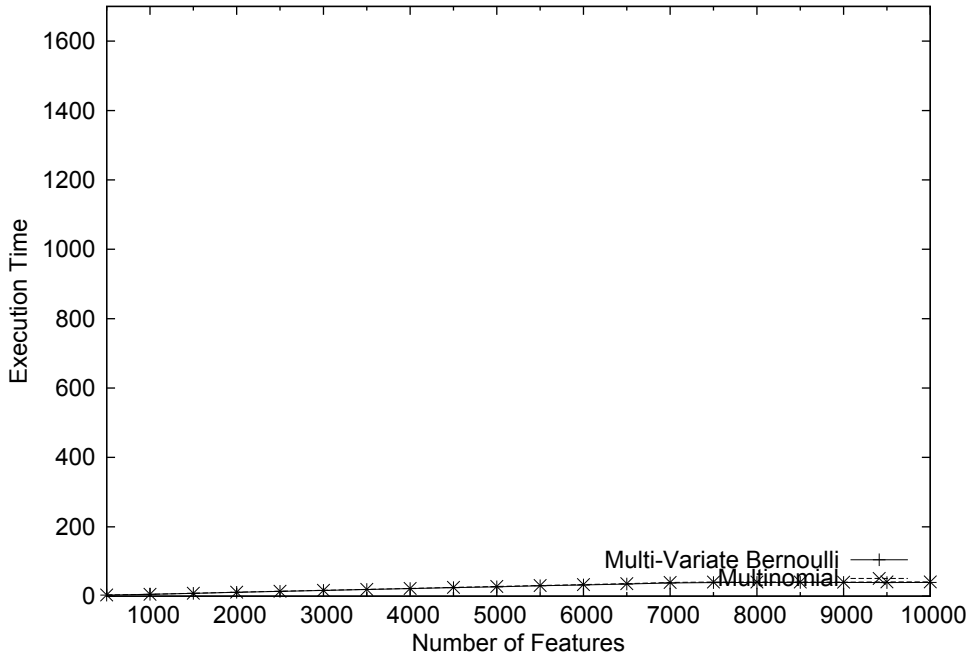


(b) Categorization time (in seconds)

Figure 5.4: Test results for `reuters_stemmed`



(a) Accuracy



(b) Categorization time (in seconds)

Figure 5.5: Test results for `webkb_stemmed`

`reuters_stemmed`, and `webkb_stemmed` datasets respectively. All three plots show that the multinomial event model achieves better accuracy than the multi-variate Bernoulli event model, with difference approximately 0.1.

Figures 5.3 (b), 5.4 (b), and 5.5 (b) show the categorization time of the `20news_stemmed`, `reuters_stemmed`, and `webkb_stemmed` datasets respectively. All three plots show that the multinomial event model has approximately the same categorization time as that of the multi-variate Bernoulli event model.

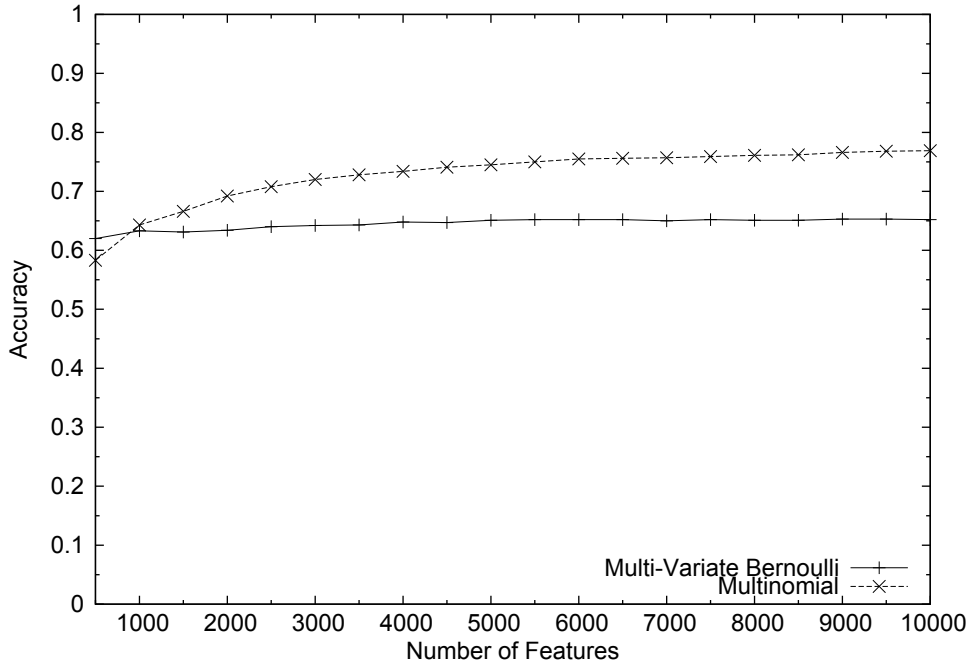
5.5 Test Results for All Terms

Figures 5.6 (a) and 5.7 (a) show the accuracy of the `20news_all_terms` and `reuters_all_terms` datasets respectively. Both of these plots show that the multinomial event model achieves better accuracy than the multi-variate Bernoulli event model, with difference approximately 0.1.

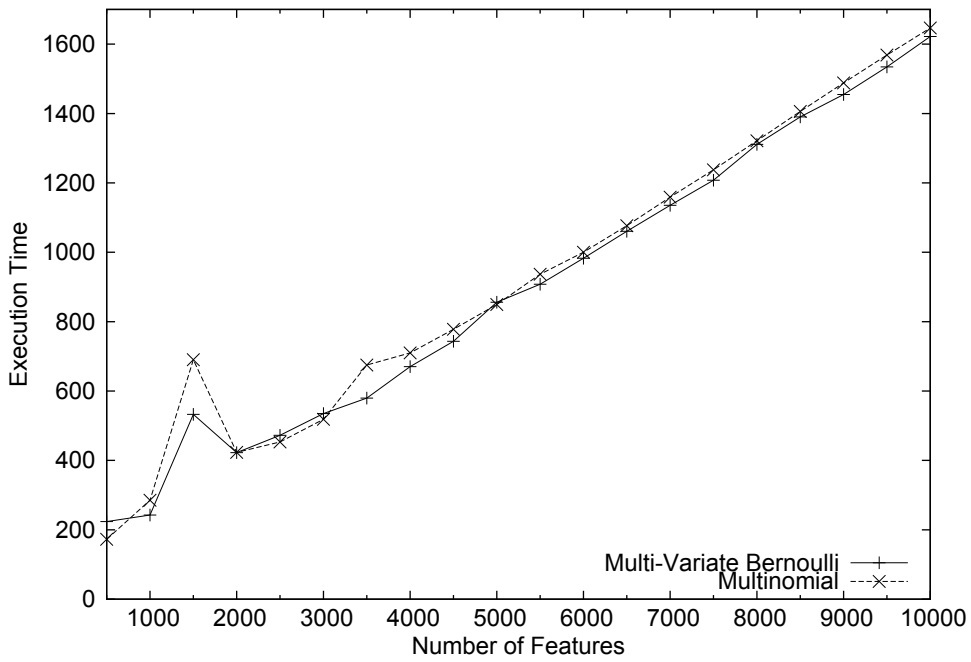
Figure 5.6 (b) and 5.7 (b) show the categorization time of the `20news_all_terms` and `reuters_all_terms` datasets respectively. Both of these plots show that the multinomial event model has approximately the same categorization time as that of the multi-variate Bernoulli event model.

5.6 Discussion

As shown in our test results, training a Naive Bayes text classifier takes a long time regardless of the event model used. This is especially true when the training set contains a large number of documents and each document contains a large number of words, such as the 20 Newsgroups dataset. However, the training time is only a one-time cost. After a Naive Bayes text classifier is trained, the text classification proceeds quickly. Both of the Reuters-21578 and WebKB test sets require less than 3 minutes of categorization time. Even categorizing the 20 Newsgroups test set with the number of features set to $10,000$ takes less than 30 minutes to finish, far less than the time it took for training. Therefore, it is feasible to use the Naive Bayes text classifier for various text classification tasks. As for the event model, we recommend multinomial because it offers approximately 0.1 better accuracy than that of multi-variate Bernoulli.

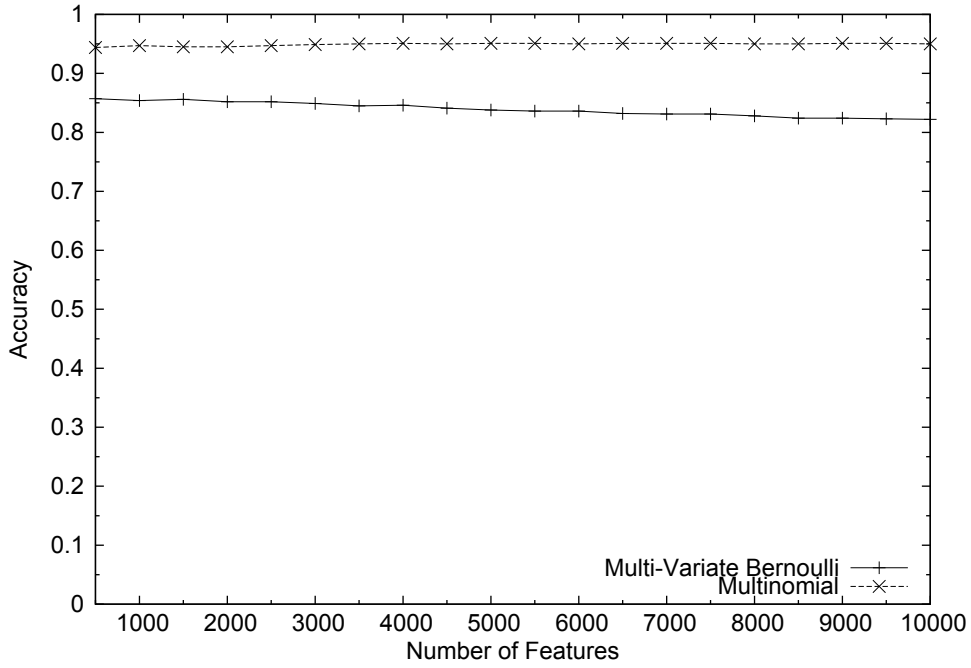


(a) Accuracy

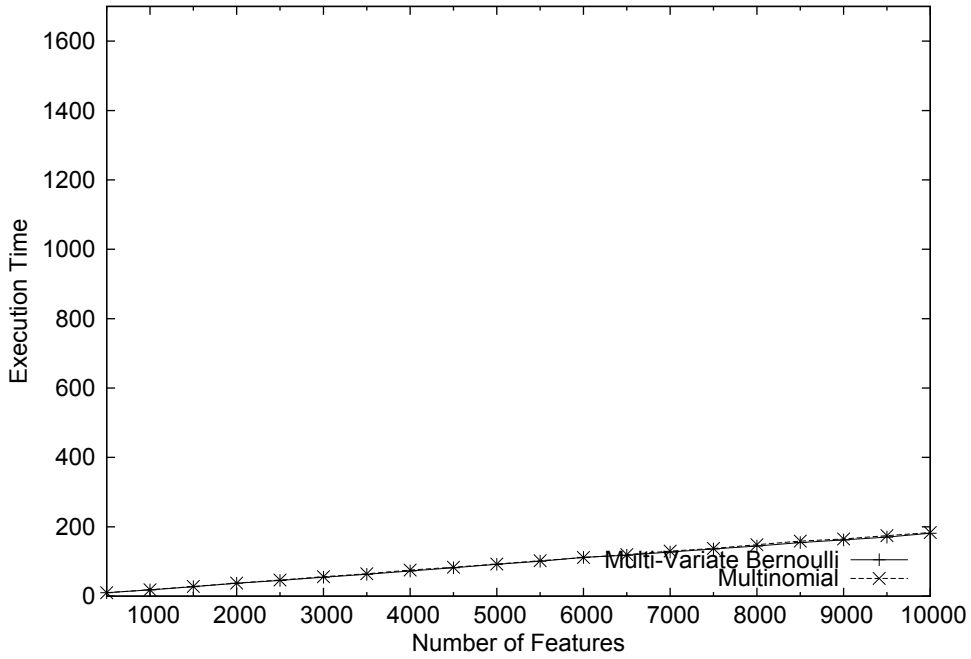


(b) Categorization time (in seconds)

Figure 5.6: Test results for `20news_all_terms`



(a) Accuracy



(b) Categorization time (in seconds)

Figure 5.7: Test results for `reuters_all_terms`

Chapter 6

Paper Categorization

Having demonstrated the capabilities of the Naive Bayes text classifier, we now turn our focus to paper categorization. Paper categorization is an application of text classification that can benefit many researchers in their literature survey efforts. For example, a literature survey usually starts with a few papers that a researcher may find relevant to the topic which he or she is investigating. Obviously, these few papers alone do not provide enough coverage of the topic under investigation. The researcher therefore collects more papers by going through the references of the aforementioned papers. However, typical references are not organized according to their categories. In order to determine whether the references are relevant to the topic under investigation, the researcher has to manually categorize the references according to their topics. In this chapter, we are going to show that

1. paper categorization can be automated by the use of the Naive Bayes text classifier,
2. paper abstracts alone are enough for achieving a high degree of accuracy, and
3. the Iris GUI can help the researchers to easily locate the papers of interest.

6.1 Collecting Papers

For demonstration purposes, papers that are categorized as `Machine learning` by the 2012 ACM Computing Classification System [9]—ACM CCS for short—are collected. ACM CCS is the classification system used in the ACM Digital Library [10]. Paper categorization is performed by a group of 120 ACM volunteers, each of whom is an

expert in certain research fields. Under ACM CCS, paper categories are organized into a tree structure. For example, category `Machine learning` has a parent category called `Computing methodologies`. Category `Computing Methodologies` in turn has a parent category called `CCS`, which also serves as the root of the category tree.

6.1.1 Training Set

A total of 115 paper abstracts collected from the ACM Digital Library are used as the training set. Each of these papers belongs to a subcategory under category `Machine learning`. Figure 6.1 shows the category tree. Each tree node contains the name of a category and the number of papers that belong to the category. For example, category `Supervised learning by classification` has 10 papers. As another example, category `Learning paradigms` has 29 papers. This is derived by summing the paper counts for the three child categories: `Supervised learning`, `Unsupervised learning`, and `Reinforcement learning`.

Because Iris does not support hierarchical categories, the training set is converted into a single-labelled dataset by using only the leaf categories. As a result, the training set is labelled with a total of 12 categories.

6.1.2 Test Set

A total of 49 paper abstracts collected from the references section of Cardoso-Cachopo's Ph.D. dissertation [1] are used as the test set. As suggested by the paper titles, all of these papers belong to category `Machine learning`. It is up to the text classifier to determine which of the 12 categories each of these papers belong to.

6.2 Setup

Before a researcher can make use of the Iris GUI, he or she must setup the environment as explained in the following subsections. Again, the machine which we used to run Iris is a HP Pavilion PC with an AMD Athlon 64 X2 Dual Core Processor 5200+ and 2 gigabytes of RAM.

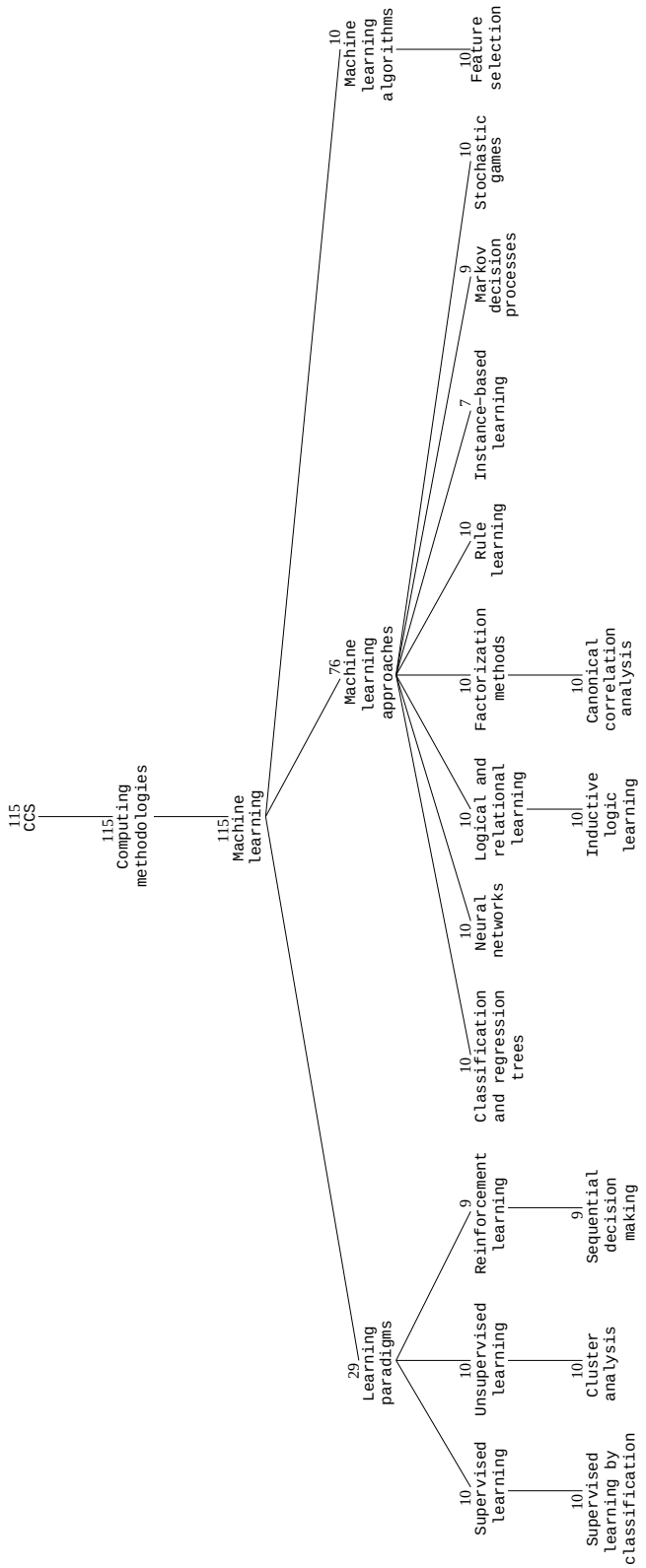


Figure 6.1: Hierarchical categories extracted from ACM CCS

6.2.1 Training Iris

As mentioned in Chapter 5, multinomial is the event model of choice because it offers better accuracy. We therefore trained Iris by using the multinomial event model. The training proceeds quickly as it only took 77.360 seconds to finish.

6.2.2 Performance Evaluation

It is not easy to determine the accuracy of paper categorization given that the test set is unlabelled. However, a rough estimate can be obtained by running the text classifier—already trained with the training set—against the training set. With this approach, a high degree of accuracy is expected since the text classifier is run against the dataset in which it was trained. Figure 6.2 shows the test results. The accuracy is satisfactory as it is above 0.9. The categorization time is also satisfactory as it is within 5 seconds.

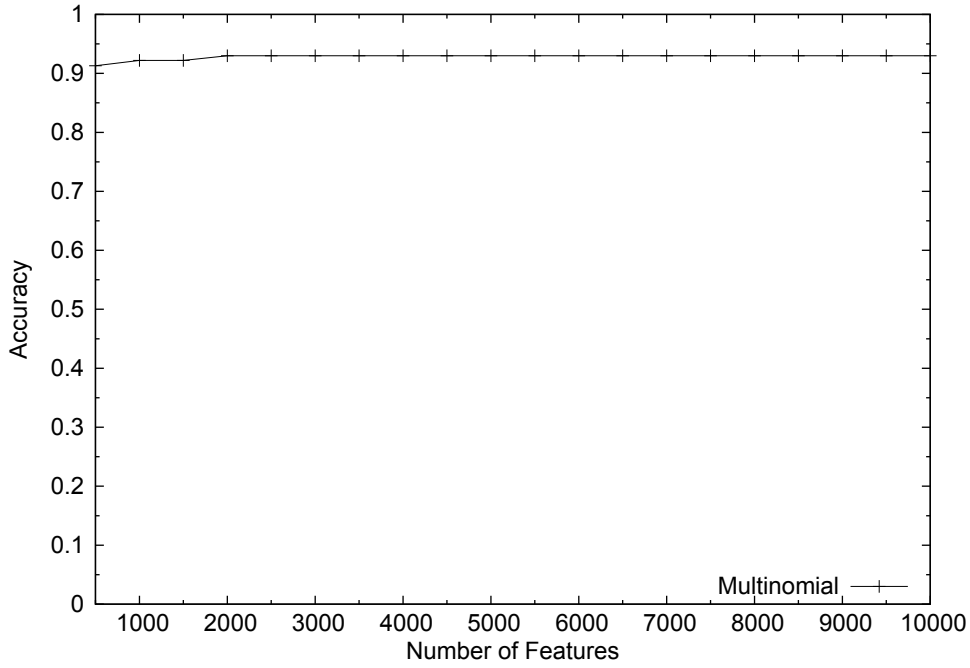
The accuracy is not perfect because all the paper abstracts are of category `Machine Learning`. As a result, some paper abstracts of different sub-categories may use the same set of distinct words. In such cases, it is difficult even for a human expert to distinguish one category from the other.

6.2.3 Applying Iris

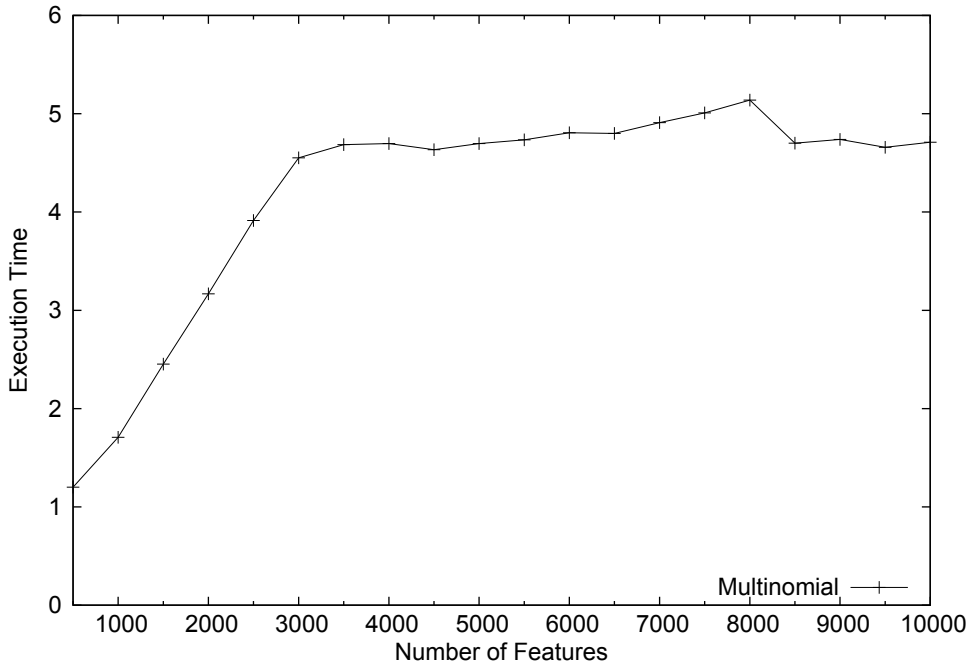
As shown in Figure 6.2, the accuracy is the highest when the number of features used is 2000. We therefore ran the Iris GUI with the number of features set to 2000 and the event model set to multinomial. After bringing up the GUI, we clicked on the `Open Directory...` menu item located under the `File` menu in order to open the directory that contains the test set. At this point, every paper in the test set is categorized. The researcher can now use the GUI to begin the paper searching process.

6.3 Searching Papers

The Iris GUI supports three different search methods as explained in the following subsections.



(a) Accuracy



(b) Categorization time (in seconds)

Figure 6.2: Test results for `paper_abstracts`

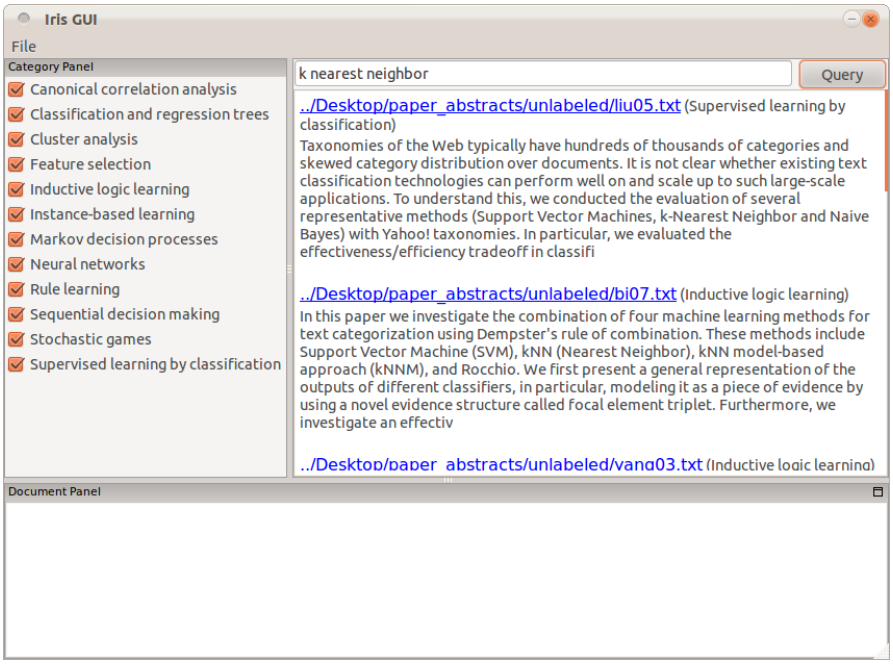


Figure 6.3: Screenshot for searching by query string

6.3.1 Searching by a Query String

This method is used when the researcher knows the keywords but not the categories of the papers under search. He or she therefore selects all the categories, types the keywords in the query field, and clicks the `Query` button. Figure 6.3 shows the search results for query `k nearest neighbor`, a text classification algorithm. The search results contain papers from three categories: `Supervised learning by classification`, `Inductive logic learning`, and `Feature selection`.

6.3.2 Searching by Categories

This method is used when the researcher knows the categories but not the keywords of the papers under search. He or she therefore selects the desired categories, leaves the query field blank, and clicks the `Query` button. Figure 6.4 shows the search results for categories `Classification and regression trees`, `Cluster analysis`, `Feature selection`, and `Supervised learning by classification`.

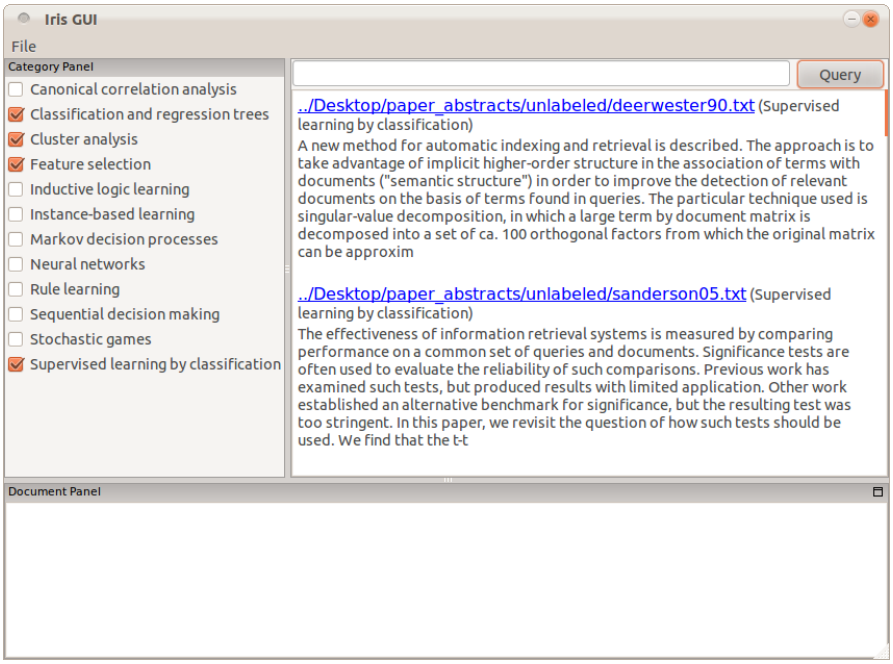


Figure 6.4: Screenshot for searching by category

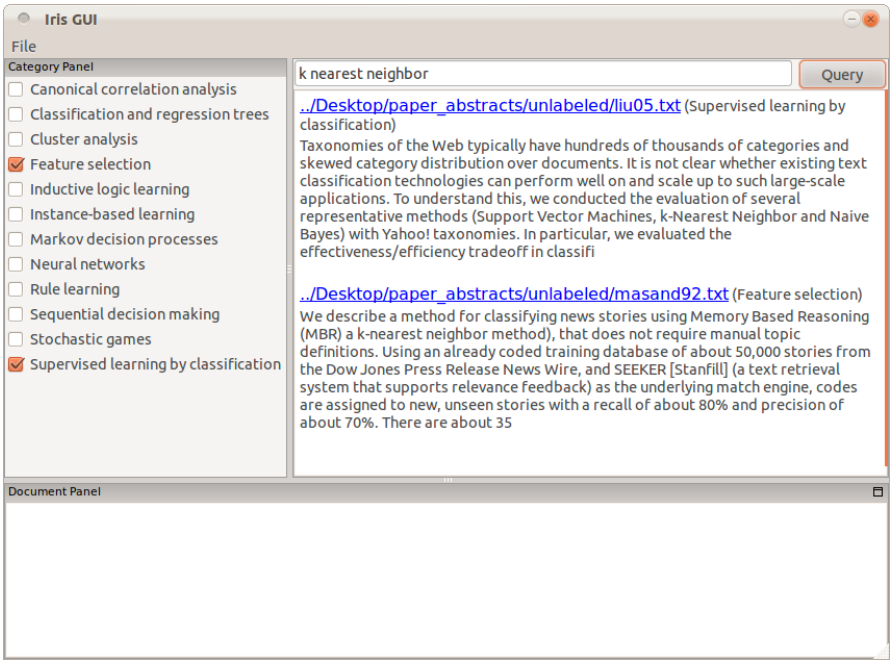


Figure 6.5: Screenshot for searching by query string and category

6.3.3 Searching by a Query String and Categories

This method is used when the researcher knows both the keywords and the categories of the papers under search. This usually occurs after searching by a query string returns papers from various categories. In order to narrow down the search, he or she then selects the desired categories, types the keywords in the query field, and clicks the `Query` button. Figure 6.5 shows the search results for query `k nearest neighbor` and categories `Feature selection` and `Supervised learning by classification`. Note that the search results presented in this figure is a subset of the search results presented in Figure 6.3. This is so because category `Inductive logic learning` is left out from the search.

6.4 Discussion

Paper categorization is a difficult problem because a typical paper tends to use concepts from many different fields. Even an expert takes a long time to categorize a paper as he or she needs to read the abstract, the introduction, and the conclusion in order to accurately identify the category of the paper. This chapter demonstrated that paper categorization can be automated by the use of the Naive Bayes text classifier. With the test results, we were able to show that the Naive Bayes text classifier is quick to train and apply, highly accurate, and highly efficient because it needs only the paper abstracts. In addition, the power of the Naive Bayes text classifier is made available to the researchers by the Iris GUI. With the screenshots, we were able to show how easy it is to locate the papers of interest through the use of the query field and the category checkboxes.

As a side note, search engines may benefit from the searching mechanism that we have just shown. For example, searching for people's profile is a tedious process for most cases. This is so because there are many people with exactly the same name. As a result, the user has to manually go through the search results to see which person is the one he or she is looking for. It would be beneficial if search engines are able to categorize people's profile into different professions so that the user can quickly determine their relevance at a glance.

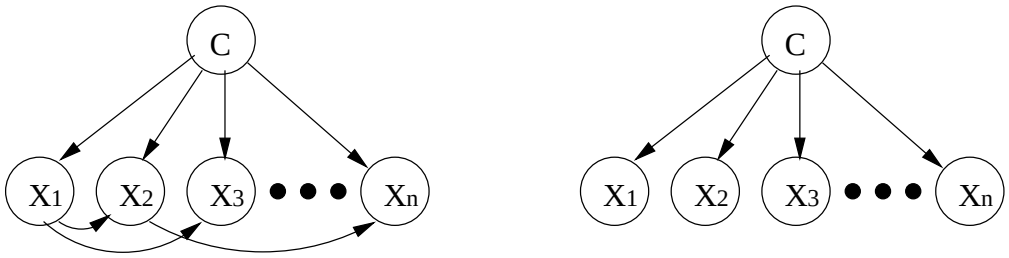
Chapter 7

Related Work

Text classification has seen many successful applications over the past forty years. Some of the most notable applications are ICD9 codes assignment for inpatient discharge summaries [11], page clustering and ranking for search engines [12, 13, 14, 15], authorship attribution [16], text genre classification [16], topic detection [16], and e-mail spam filters [17]. While many text classification algorithms exist, Bayesian's simplicity and accuracy makes it popular in the text classification literature. The following sections present the variants of Bayesian text classifiers, the algorithms for feature selection, and the metrics used for performance evaluation.

7.1 Variants of Bayesian Text Classifiers

With Bayesian text classification, a document is represented as a feature vector $X_1, X_2, X_3, ..., X_n$. From the feature vector we can then derive the score of the document belonging to category C, denoted as $score(C)$. $score(C)$ is difficult to calculate in practice because of the dependencies that exist between features as shown in Figure 7.1 (a). To make the calculation easier, the features are assumed independent of each other and the resulting text classifier is termed Naive Bayes. Figure 7.1 (b) shows the Bayesian network with the independence assumption. It is obvious that such an assumption is unrealistic to real world scenarios since many words have dependencies. For example, the word machine tends to precede the word learning in the machine learning literature. By intuition, any text classifier that ignores feature dependencies ought to have low accuracy. However, Naive Bayes performs quite well according to McCallum et al. [2] and Sahami et al. [17]. Below we present several



(a) Dependent features (b) Independent features

Figure 7.1: Bayesian networks

variants of Naive Bayes text classifiers.

7.1.1 Multi-Variate Bernoulli Model

The multi-variate Bernoulli event model is commonly used in Naive Bayes text classifiers as presented by Eyheramendy et al. [18], Lewis [19], McCallum et al. [2], and Sahami et al. [17]. With this event model, each element in the feature vector denotes presence or absence of a word in a document.

7.1.2 Multinomial Model

The multinomial event model is also commonly used in Naive Bayes text classifiers as presented by Eyheramendy et al. [18], Joachims [20], Lewis [19], and McCallum et al. [2]. With this event model, each element in the feature vector denotes the number of occurrences of a word in a document.

7.1.3 Restricted Bayesian Network

As mentioned before, Naive Bayes simplifies the calculation of $score(C)$ by making the independence assumption. Although it achieves respectable accuracy, Friedman et al. [21], Peng et al. [16], and Sahami [22] have attempted to increase its accuracy even further by relaxing the independence assumption. Because capturing the full dependencies is too expensive in terms of machine execution time, they simplify the problem by capturing only limited dependencies. This is achieved by allowing each word to have at most one dependency to another word. The experimental results show that the accuracy improves remarkablely when limited dependencies are incorporated. They do not show the machine execution time with respect to Naive Bayes, however.

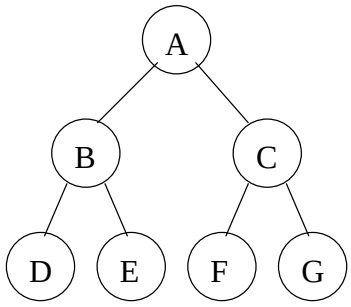


Figure 7.2: Category hierarchy

7.1.4 Hierarhical Classification

The need for hierarchical classification arises when the categories are organized into a hierarchy. Such is the case for 2012 ACM Computing Classification System as presented in Chapter 6. Koller et al. [23] demonstrated how hierarchical classification can be achieved by using a series of Naive Bayes text classifiers, one for each category with sub-categories. To illustrate, Figure 7.2 shows an example category hierarchy with seven categories. Text classification starts from the root category `A`. A Naive Bayes text classifier is applied to categorize documents into either `B` or `C`. For each document that is categorized as category `B`, another Naive Bayes text classifier is applied to further categorize documents into either `D` or `E`. With this approach, text classification continues until every document is labelled with a leaf category.

7.1.5 Multi-Label Classification

As mentioned in Chapter 2, a Naive Bayes text classifier determines a document's category by calculating the score for each category. The category with the highest score is the document's category. With this approach, Naive Bayes supports only single-label classification. Wei et al. [14], however, demonstrated how Naive Bayes can be modified to support multi-label classification. They accomplished this by setting a threshold. Any category with score exceeding the threshold is determined as the document's category.

7.2 Feature Selection

Text classification is a time-consuming process. This is especially true when the documents under categorization have large vocabulary. On the other hand, long categorization time can be avoided if the vocabulary size is reduced. Such a reduction can be achieved by the use of feature selection, an algorithm for picking those words that have more say in determining the category of a document. To our knowledge, there exists two algorithms for feature selection. The first algorithm—used by Koller et al. [23] and Sahami et al. [17]—makes use of Zipf's law to determine which words should be left out. The second algorithm—used by Joachims [20], McCallum et al. [2], and Sahami et al. [17]—relies on mutual information to determine which words should be used. Mutual information provides a measure of dependency between a word and all categories. The higher the mutual information, the more useful a word is and vice versa.

7.3 Evaluation Metrics

7.3.1 Accuracy

Accuracy is calculated by dividing the number of correctly categorized documents by the total number of documents. It is used by Koller et al. [23], McCallum [24], Peng et al. [16], and Sahami [22].

7.3.2 Precision and Recall

Given a category C, let X be the set of documents that are of category C and let Y be the set of documents that are deemed by the text classifier to be of category C. Precision is defined as $|X \cap Y|/|Y|$, which is the portion of the documents that are deemed as category C by the text classifier are indeed of category C. Recall is defined as $|X \cap Y|/|X|$, which is the portion of the documents that are of category C are deemed as category C by the text classifier. Precision and recall are used by Lewis et al. [25] and Sahami et al. [17].

More formally, precision and recall can be explained in terms of prediction and actual according to Olson [26]. Predicted is either positive p or negative n. Actual is true t when the prediction is correct and false f otherwise. They together yield the following four scenarios:

- *tp*: occurs when prediction is positive and the prediction is true.
- *fp*: occurs when prediction is positive but the prediction is false.
- *tn*: occurs when prediction is negative and the prediction is true.
- *fn*: occurs when prediction is negative but the prediction is false.

With these definitions, precision and recall can be expressed as:

$$Precision = \frac{tp}{tp + fp} \tag{7.1}$$

$$Recall = \frac{tp}{tp + fn} \tag{7.2}$$

Chapter 8

Conclusions

Paper categorization is essential in the literature survey efforts as it facilitates easy paper searching. It is an instance of a more general problem called text classification which has been studied extensively over the past forty years. Among many text classification algorithms, Naive Bayes is widely used because of its accuracy, speed, and simplicity. Through our experiments, we were able to confirm the advantages of using Naive Bayes as it is reasonablely accurate, fast in terms of categorization time, and relatively easy to implement. As a result, we chose Naive Bayes for paper categorization. Below we summarize the contributions of this thesis and provide a few directions for future research.

8.1 Contributions

First, a prototype that facilitates the evaluation of Naive Bayes against datasets of choice is introduced. The prototype offers five commands. Three of the commands—`frequency.py`, `mutual_info.py`, and `probability.py`—are used for training the text classifier, each of which calculates the statistics of a training set and stores the results into a database. Both `mutual_info.py` and `probability.py` allow the user to specify the event model through the command line argument *model*. The other two commands—`categorize.py` and `plot.py`—are used for performance evaluation. `categorize.py` categorizes a test set using the statistics stored in the database and generates a report containing the accuracy and categorization time. It allows the user to specify the number of features through the command line argument *num_features*. `plot.py` uses the report to generate plots, one for accuracy with respect to number

of features and the other for categorization time with respect to number of features.

Second, the results of running Naive Bayes against a number of popular datasets are presented. In Chapter 5, we ran Naive Bayes against the 20 Newsgroups, the Reuters-21578, and the WebKB datasets using both event models. The results show that the multinomial model achieves approximately 0.1 better accuracy than that of the multi-variate Bernoulli model. On the other hand, both models have similar training and categorization time. In particular, the categorization time is always significantly shorter than the training time, especially when the number of features used is small. Since a Naive Bayes text classifier is likely to be trained once and applied many times, the fact that the training time is long should not pose a significant problem.

Third, a GUI for assisting users with categorizing documents of unknown categories and locating the documents of interest is proposed. To use the GUI, the user first trains the text classifier using the aforementioned commands and then invokes `gui.py` to bring up the GUI. Document categorization is accomplished by opening a directory that contain a set of documents that the user wishes to categorize. Document searching is accomplished by entering a query string in the query field, selecting all categories of interest, or a combination of both.

Finally, the steps of paper categorization and searching through GUI are demonstrated. In Chapter 6, we collected 115 paper from the ACM Digital Library and used them as the training set. The classification system used by ACM is hierarchical which is incompatible with the Naive Bayes text classifier. We therefore converted the category hierarchy into a flat structure by considering only the leaf categories. With the category structure converted, we then trained the text classifier using the training set. Because the performance of Naive Bayes varies on different datasets, it is important to test the text classifier's performance. In this case, the text classifier is applied to the training set. The test results show that the text classifier achieves high accuracy, an expected outcome since the same dataset was used for training. Finally, we applied the text classifier to papers that we extracted from the references section of Cardoso-Cachopo's Ph.D. dissertation [1] so that paper searching can proceed. Paper searching is facilitated by the GUI which offers a query field and a checkbox for each category. Through the GUI, one can search a paper or a set of papers by a query string, a set of categories, or a combination of both. With this approach, the search results contain only a handful of papers which the user can decide whether they are of interest at a glance.

8.2 Future Work

There are plenty of directions for future research. One possible direction is improving the accuracy of Naive Bayes by considering the dependencies between the features. We expect the accuracy to increase slightly as the Naive Bayes is already performing so well. Another possible direction is speeding up the text classification through code optimization. Since the focus of this thesis is paper categorization, we consider the following more important than the rest.

8.2.1 Hierarchical Classification

A typical research topic usually has a number of subtopics. It is only natural that the classification system used for papers is hierarchical. As such, our prototype can be improved by incorporating hierarchical classification. The approach introduced by Koller et al. [23] seems promising in this regard.

8.2.2 Multi-Label Classification

Another unique aspect about paper categorization is that typical papers tend to use concepts from many research topics. It is not uncommon that a paper is labelled with two or more categories by human experts. As such, our prototype can be improved by incorporating multi-label classification. The approach introduced by Wei et al. [14] seems promising in this regard.

Appendix A

Database Schema

```
create table document (
        id int not null,
        primary key (id)
);

create table category (
        id int not null,
        category varchar(90) not null,
        probability float not null,
        primary key (id)
);

create table document_category (
        document_id int not null,
        category_id int not null,
        foreign key (document_id) references document(id),
        foreign key (category_id) references category(id)
);

create table word (
        id int not null,
        word varchar(90) not null,
        mutual_info float not null,
        primary key (id)
);

create table inverted_index (
        word_id int not null,
        document_id int not null,
        frequency int not null,
        foreign key (word_id) references word(id),
        foreign key (document_id) references document(id),
        primary key (word_id,document_id)
);
```

```
create table word_category_probability (
        word_id int not null,
        category_id int not null,
        probability float not null,
        foreign key (word_id) references word(id),
        foreign key (category_id) references category(id),
        primary key (word_id,category_id)
);
```

Appendix B

Python Modules and Iris Commands

B.1 util.py

```
import MySQLdb

class DB:
        def __init__(self,database,user,password):
                '''
                Purpose
                        Connect to database authenticated by user and password.
                        Throws an exception when connect fails.
                '''
                self.conn = MySQLdb.connect(host='localhost',db=database, \
                 user=user,passwd=password)
                self.category_id = 1
                self.word_id = 1

### methods for frequency.py

        def get_category_id(self,c):
                '''
                Purpose
                        If category c exists:
                                return its id.
                        Else:
                                return 0.
                Precondition
                        c is a string.
                '''
                category_id = 0
                statement = "select id from category where category = '%s'"
```

```
        statement = statement % (c)
        cursor = self.conn.cursor()
        cursor.execute(statement)
        rows = cursor.fetchall()
        for row in rows:
                category_id = int( row[0] )
        return category_id

def insert_category(self,c):
        '''
        Purpose
                Insert category c.
        Precondition
                c is a string.
        '''
        statement = "insert into category (id,category,probability) "+\
         "values (%d,'%s',0.0)"
        statement = statement % (self.category_id,c)
        cursor = self.conn.cursor()
        cursor.execute(statement)
        self.conn.commit()

        self.category_id += 1

def get_word_id(self,w):
        '''
        Purpose
                If word w exists:
                        return its id.
                Else:
                        return 0.
        Precondition
                w is a string.
        '''
        # escape single quote(s)
        w = w.replace('\'','\\\'')

        word_id = 0
        statement = "select id from word where word = '%s'"
        statement = statement % (w)
        cursor = self.conn.cursor()
        cursor.execute(statement)
        rows = cursor.fetchall()
        for row in rows:
                word_id = int( row[0] )
        return word_id

def insert_word(self,w):
        '''
        Purpose
                Insert word w.
        Precondition
```

```
                w is a lower-case string.
        '''
        # escape single quote(s)
        w = w.replace('\'','\\\'')

        statement = "insert into word (id,word,mutual_info) " + \
         "values (%d,'%s',0.0)"
        statement = statement % (self.word_id,w)
        cursor = self.conn.cursor()
        cursor.execute(statement)
        self.conn.commit()

        self.word_id += 1

def insert_document(self,did):
        '''
        Purpose
                Insert document did.
        Precondition
                did is an integer.
        '''
        statement = "insert into document (id) values (%d)"
        statement = statement % (did)
        cursor = self.conn.cursor()
        cursor.execute(statement)
        self.conn.commit()

def insert_document_category(self,cid,did):
        '''
        Purpose
                Insert a document category of category ID cid and
                 document ID did.
        Precondition
                cid and did are integers.
        '''
        statement = "insert into document_category " + \
         "(category_id,document_id) values (%d,%d)"
        statement = statement % (cid,did)
        cursor = self.conn.cursor()
        cursor.execute(statement)
        self.conn.commit()

def insert_inverted_index(self,wid,did,f):
        '''
        Purpose
                Insert an inverted index of word ID wid, document ID
                 id, and frequency f.
        Precondition
                wid, did, and f are integers.
        '''
        statement = "insert into inverted_index " + \
         "(word_id,document_id,frequency) values (%d,%d,%d)"
```

```
                statement = statement % (wid,did,f)
                cursor = self.conn.cursor()
                cursor.execute(statement)
                self.conn.commit()

### methods for mutual_info.py

        def get_word_dict(self,num_words=0):
                '''
                Purpose
                        Return a dictionary of words where the keys are words
                         and the values are word IDs.
                        If num_words == 0:
                                Return all words.
                        Else:
                                Return num_words words.
                '''
                word_dict = {}

                if num_words == 0:
                        statement = "select id,word from word"
                else:
                        statement = "select id,word from word " + \
                         "order by mutual_info desc limit %d"
                        statement = statement % (num_words)

                cursor = self.conn.cursor()
                cursor.execute(statement)
                rows = cursor.fetchall()
                for row in rows:
                        word_dict[ row[1] ] = row[0]
                return word_dict

        def get_category_dict(self):
                '''
                Purpose
                        Return a dictionary of categories where the keys are
                         categories and the values are category IDs.
                '''
                category_dict = {}
                statement = "select id,category from category"
                cursor = self.conn.cursor()
                cursor.execute(statement)
                rows = cursor.fetchall()
                for row in rows:
                        category_dict[ row[1] ] = row[0]
                return category_dict

        def get_num_documents_of_word(self,wid):
                '''
                Purpose
                        Return the number of documents that have word of ID
```

```
                wid.
        Precondition
                wid is an integer.
        '''
        num_documents = 0
        statement = "select count(ii.document_id) " + \
         "from inverted_index ii " + \
         "where ii.word_id = %d"
        statement = statement % (wid)
        cursor = self.conn.cursor()
        cursor.execute(statement)
        rows = cursor.fetchall()
        for row in rows:
                if row[0] != None:
                        num_documents = int( row[0] )
        return num_documents

def get_num_documents_of_category(self,tid):
        '''
        Purpose
                Return the number of documents that have category of ID
                 tid.
        Precondition
                tid is an integer.
        '''
        num_documents = 0
        statement = "select count(dc.document_id) " + \
         "from document_category dc " + \
         "where dc.category_id = %d"
        statement = statement % (tid)
        cursor = self.conn.cursor()
        cursor.execute(statement)
        rows = cursor.fetchall()
        for row in rows:
                if row[0] != None:
                        num_documents = int( row[0] )
        return num_documents

def get_num_documents_of_word_category(self,wid,tid):
        '''
        Purpose
                Return the number of documents that have word of ID wid
                 and category of ID tid.
        Precondition
                wid and tid are integers.
        '''
        num_documents = 0
        statement = "select count(dc.document_id) " + \
         "from inverted_index ii, document_category dc " + \
         "where ii.document_id = dc.document_id " + \
         "and ii.word_id = %d " + \
         "and dc.category_id = %d"
```

```
        statement = statement % (wid,tid)
        cursor = self.conn.cursor()
        cursor.execute(statement)
        rows = cursor.fetchall()
        for row in rows:
                if row[0] != None:
                        num_documents = int( row[0] )
        return num_documents

def get_num_documents(self):
        '''
        Purpose
                Return the number of documents in database.
        '''
        num_documents = 0
        statement = "select count(*) " + \
         "from document"
        cursor = self.conn.cursor()
        cursor.execute(statement)
        rows = cursor.fetchall()
        for row in rows:
                if row[0] != None:
                        num_documents = int( row[0] )
        return num_documents

def update_category(self,cid,p):
        '''
        Purpose
                Update category of ID cid with probability p.
        Precondition
                cid is an integer.
                p is a float.
        '''
        statement = "update category set probability = %f " + \
         "where id = %d"
        statement = statement % (p,cid)
        cursor = self.conn.cursor()
        cursor.execute(statement)
        self.conn.commit()

def update_word(self,wid,m):
        '''
        Purpose
                Update word of ID wid with mutual information m.
        Precondition
                wid is an integer.
                m is a float.
        '''
        statement = "update word set mutual_info = %f " + \
         "where id = %d"
        statement = statement % (m,wid)
        cursor = self.conn.cursor()
```

```
                cursor.execute(statement)
                self.conn.commit()

### methods for probability.py

        def get_word_category_frequency(self,wid,tid):
                '''
                Purpose
                        Return the frequency of word of ID wid and category of
                         ID tid.
                Precondition
                        wid and tid are integers.
                '''
                frequency = 0
                statement = "select sum(ii.frequency) " + \
                 "from inverted_index ii, document_category dc " + \
                 "where ii.document_id = dc.document_id " + \
                 "and ii.word_id = %d " + \
                 "and dc.category_id = %d"
                statement = statement % (wid,tid)
                cursor = self.conn.cursor()
                cursor.execute(statement)
                rows = cursor.fetchall()
                for row in rows:
                        if row[0] != None:
                                frequency = int( row[0] )
                return frequency

        def get_category_frequency(self,tid):
                '''
                Purpose
                        Return the frequency of category of ID tid.
                Precondition
                        tid is an integer.
                '''
                frequency = 0
                statement = "select sum(ii.frequency) " + \
                 "from inverted_index ii, document_category dc " + \
                 "where ii.document_id = dc.document_id " + \
                 "and dc.category_id = %d"
                statement = statement % (tid)
                cursor = self.conn.cursor()
                cursor.execute(statement)
                rows = cursor.fetchall()
                for row in rows:
                        if row[0] != None:
                                frequency = int( row[0] )
                return frequency

        def get_word_frequency(self,wid):
                '''
                Purpose
```

```
                Return the frequency of word of ID wid.
            Precondition
                wid is an integer.
            '''
            frequency = 0
            statement = "select sum(ii.frequency) " + \
             "from inverted_index ii " + \
             "where ii.word_id = %d"
            statement = statement % (wid)
            cursor = self.conn.cursor()
            cursor.execute(statement)
            rows = cursor.fetchall()
            for row in rows:
                if row[0] != None:
                    frequency = int( row[0] )
            return frequency

    def get_frequency(self):
            '''
            Purpose
                Return the total word count.
            '''
            frequency = 0
            statement = "select sum(ii.frequency) " + \
             "from inverted_index ii"
            cursor = self.conn.cursor()
            cursor.execute(statement)
            rows = cursor.fetchall()
            for row in rows:
                if row[0] != None:
                    frequency = int( row[0] )
            return frequency

    def insert_word_category_probability(self,wid,tid,p):
            '''
            Purpose
                Insert a log(Pr(word|category)) p for word of ID wid and
                 category of ID tid.
            Precondition
                wid and tid are integers.
                p is a float.
            '''
            statement = "insert into word_category_probability " + \
             "(word_id,category_id,probability) " + \
             "values(%d,%d,%f)"
            statement = statement % (wid,tid,p)
            cursor = self.conn.cursor()
            cursor.execute(statement)
            self.conn.commit()

### methods for bayes.py
```

```
        def get_category_probabilities(self):
                '''
                Purpose
                        Return a dictionary of log(Pr(category)) where the keys
                         are category IDs and the values are log(Pr(category)).
                '''
                P = {}
                statement = "select id,probability from category"
                cursor = self.conn.cursor()
                cursor.execute(statement)
                rows = cursor.fetchall()
                for row in rows:
                        P[ row[0] ] = row[1]
                return P

        def get_word_category_probabilities(self):
                '''
                Purpose
                        Return a dictionary of log(Pr(word|category)) where the
                         keys are word and category IDs and the values are
                         log(Pr(word|category)).
                '''
                P = {}
                statement = "select word_id,category_id,probability " + \
                 "from word_category_probability"
                cursor = self.conn.cursor()
                cursor.execute(statement)
                rows = cursor.fetchall()
                for row in rows:
                        word_id,category_id,probability = row[0:3]
                        if not P.has_key(word_id):
                                P[word_id] = {}
                        P[word_id][category_id] = probability
                return P

        def close(self):
                '''
                Purpose
                        Disconnect from database.
                        Throws an exception when disconnect fails.
                '''
                self.conn.close()

def get_word_frequencies(document_body):
        '''
        Purpose
                Return a dictionary where the keys are words and the values are
                 word frequencies by parsing document_body.
        Precondition
                document_body is a string.
        '''
        word_frequencies = {}
```

```
        for word in document_body.split():
                # convert word to lower-case
                word = word.lower()

                # strips trailing punctuation
                if not 'a' <= word[-1] <='z':
                        word = word[:-1]

                # determine whether word is valid
                valid_word = True
                for c in word:
                        if not 'a' <= c <= 'z' and c not in ['\'','-']:
                                valid_word = False

                # add word to dictionary if word is valid and not stop word
                if valid_word:
                        if word_frequencies.has_key(word):
                                frequency = word_frequencies[word]
                                word_frequencies[word] = frequency+1
                        else:
                                word_frequencies[word] = 1

        return word_frequencies
```

B.2 frequency.py

```
import sys
import util
import MySQLdb
import glob

if __name__ == '__main__':
        # check command line arguments
        if len(sys.argv) != 5:
                print 'Usage: python frequency.py ' + \
                 '<database> <user> <password> <train_dir>'
                sys.exit(-1)

        # connect to database
        try:
                db = util.DB(sys.argv[1],sys.argv[2],sys.argv[3])
        except MySQLdb.Error,e:
                print 'Error %d: %s' % (e.args[0],e.args[1])
                sys.exit(-1)

        # build an inverted index for documents in train_dir
        document_id = 1
        for document_path in glob.glob( sys.argv[4]+'/*/*' ):
                # open document file for reading
```

```
        try:
                file_handle = open(document_path,'r')
        except IOError,e:
                print 'I/O Error %d: %s' % (e.args[0],e.args[1])
                sys.exit(-1)

        # read words from document file
        word_frequencies = util.get_word_frequencies( \
         file_handle.read())
        file_handle.close()

        # extract category from document path
        category = document_path.split('/')[-2]

        # save document
        db.insert_document(document_id)

        # save category
        category_id = db.get_category_id(category)
        if category_id == 0: # category not exist
                db.insert_category(category)
                category_id = db.get_category_id(category)

        # save document category relation
        db.insert_document_category(category_id,document_id)

        # save word frequencies
        for word in word_frequencies.keys():
                # save word
                word_id = db.get_word_id(word)
                if word_id == 0: # word not exist
                        db.insert_word(word)
                        word_id = db.get_word_id(word)

                # save inverted index
                frequency = word_frequencies[word]
                db.insert_inverted_index( \
                 word_id,document_id,frequency)

        # update document ID
        document_id += 1

# disconnect from database
try:
        db.close()
except MySQLdb.Error,e:
        print 'Error %d: %s' % (e.args[0],e.args[1])
        sys.exit(-1)
```

B.3 mutual_info.py

```
import util
import sys
import math

def get_pr_category(db,cid,M):
        '''
        Purpose
                Calculating P(category) of category cid by using event model M
                 and return the result.
        Precondition
                db is successfully returned from DB.__init__.
                cid is an integer.
                M is either 'multivariate' or 'multinomial'.
        '''
        if M == 'multivariate':
                numerator = db.get_num_documents_of_category(cid)
                denominator = db.get_num_documents()
                return float(numerator) / float(denominator)
        elif M == 'multinomial':
                numerator = db.get_category_frequency(cid)
                denominator = db.get_frequency()
                return float(numerator) / float(denominator)
        return 0.0

def get_pr_word(db,wid,M):
        '''
        Purpose
                Calculating P(word) of word wid by using event model M and
                 return the result.
        Precondition
                db is successfully returned from DB.__init__.
                wid is an integer.
                M is either 'multivariate' or 'multinomial'.
        '''
        if M == 'multivariate':
                numerator = db.get_num_documents_of_word(wid)
                denominator = db.get_num_documents()
                return float(numerator) / float(denominator)
        elif M == 'multinomial':
                numerator = db.get_word_frequency(wid)
                denominator = db.get_frequency()
                return float(numerator) / float(denominator)
        return 0.0

def get_pr_category_word(db,cid,wid,M):
        '''
        Purpose
                Calculating P(category,word) of category cid and word wid by
                 using event model M and return the result.
```

```
        Precondition
                db is successfully returned from DB.__init__.
                cid is an integer.
                wid is an integer.
                M is either 'multivariate' or 'multinomial'.
        '''
        if M == 'multivariate':
                numerator = db.get_num_documents_of_word_category(wid,cid)
                denominator = db.get_num_documents()
                return float(numerator) / float(denominator)
        elif M == 'multinomial':
                numerator = db.get_word_category_frequency(wid,cid)
                denominator = db.get_frequency()
                return float(numerator) / float(denominator)
        return 0.0

if __name__ == '__main__':
        # check command line arguments
        if len(sys.argv) != 5:
                print 'Usage: python mutual_info.py ' + \
                 '<database> <user> <password> <model>'
                sys.exit(-1)

        # connect to database
        try:
                db = util.DB(sys.argv[1],sys.argv[2],sys.argv[3])
        except MySQLdb.Error,e:
                print 'Error %d: %s' % (e.args[0],e.args[1])
                sys.exit(-1)

        # get event model
        model = sys.argv[4]
        if model not in ['multivariate','multinomial']:
                print "Error: <model> must be either " + \
                 "'multivariate' or 'multinomial'"
                sys.exit(-1)

        # get words
        word_dict = db.get_word_dict()

        # get categories
        category_dict = db.get_category_dict()

        # calculate and save mutual information for each word
        M = []
        for word_id in word_dict.values():
                mutual_info = 0.0
                for category_id in category_dict.values():
                        # calculate P(category)
                        pr_category = get_pr_category(db,category_id,model)

                        # calculate Pr(word)
```

```
                        pr_word = get_pr_word(db,word_id,model)

                        # calculate P(category,word)
                        pr_category_word = get_pr_category_word(db, \
                         category_id,word_id,model)

                        # calculate mutual information
                        if pr_category_word == 0.0:
                                continue
                        mutual_info += pr_category_word * \
                         math.log(pr_category_word / pr_category / pr_word)

                # save mutual information
                db.update_word(word_id,mutual_info)

        # disconnect from database
        try:
                db.close()
        except MySQLdb.Error,e:
                print 'Error %d: %s' % (e.args[0],e.args[1])
                sys.exit(-1)
```

B.4 probability.py

```
import util
import sys

def get_pr_word_category(db,wid,cid,M,V):
        '''
        Purpose
                Calculating P(word|category) of word wid and category cid by
                 using event model M and return the result.
        Precondition
                db is successfully returned from DB.__init__.
                cid is an integer.
                wid is an integer.
                M is either 'multivariate' or 'multinomial'.
                V is the size of vocabulary.
        '''
        if M == 'multivariate':
                numerator = 1 + db.get_num_documents_of_word_category(wid,cid)
                denominator = 2 + db.get_num_documents_of_category(cid)
                return float(numerator) / float(denominator)
        elif M == 'multinomial':
                numerator = 1 + db.get_word_category_frequency(wid,cid)
                denominator = V + db.get_category_frequency(category_id)
                return float(numerator) / float(denominator)
        return 0.0
```

```
if __name__ == '__main__':
        # check command line arguments
        if len(sys.argv) != 5:
                print 'Usage: python frequency.py ' + \
                 '<database> <user> <password> <model>'
                sys.exit(-1)

        # connect to database
        try:
                db = util.DB(sys.argv[1],sys.argv[2],sys.argv[3])
        except MySQLdb.Error,e:
                print 'Error %d: %s' % (e.args[0],e.args[1])
                sys.exit(-1)

        # get event model
        M = sys.argv[4]
        if M not in ['multivariate','multinomial']:
                print "Error: <model> must be either " + \
                 "'multivariate' or 'multinomial'"
                sys.exit(-1)

        # get words
        word_dict = db.get_word_dict()
        V = len(word_dict.keys())

        # get categories
        category_dict = db.get_category_dict()

        # calculate Pr(category)
        for category_id in category_dict.values():
                numerator = db.get_num_documents_of_category(category_id)
                denominator = db.get_num_documents()
                probability = float(numerator) / float(denominator)
                db.update_category(category_id,probability)

        # calculate Pr(word|category)
        for word_id in word_dict.values():
                for category_id in category_dict.values():
                        probability = get_pr_word_category( \
                         db,word_id,category_id,M,V)
                        db.insert_word_category_probability( \
                         word_id,category_id,probability)

        # disconnect from database
        try:
                db.close()
        except MySQLdb.Error,e:
                print 'Error %d: %s' % (e.args[0],e.args[1])
                sys.exit(-1)
```

B.5 categorize.py

```
import sys
import util
import MySQLdb
import math
import glob
from multiprocessing import Pool
import time

def categorize(document_path,PC,PWC,C,W,M):
        '''
        Purpose
                Calculate the category of the document located at document_path
                 by using PC, PWC, C, W, and M and return the category ID.
        Precondition
                document_path is the path to a document.
                PC is a dictionary of Pr(category).
                PWC is a dictionary of Pr(word|category).
                C is a dictionary of categories.
                W is a dictionary of words.
                M is an event model of either 'multivariate' or 'multinomial'.
        '''
        # open document file for reading
        try:
                file_handle = open(document_path,'r')
        except IOError,e:
                print 'I/O Error %d: %s' % (e.args[0],e.args[1])
                sys.exit(-1)

        # read document body
        document_body = file_handle.read()

        # close document file
        file_handle.close()

        # build a word frequencies dictionary
        word_frequencies = util.get_word_frequencies(document_body)

        # calculate Pr(category|words) for each category
        prs = []
        for cid in C.values():
                # retrieve Pr(category)
                pr_category = PC[cid]

                # apply Pr(category)
                pr_category_words = math.log(pr_category)

                # apply Pr(word|category)s
                for word in W.keys():
                        # get word count
```

```
                        word_count = 0
                        if word_frequencies.has_key(word):
                                word_count = word_frequencies[word]

                        # retrieve Pr(word|category)
                        pr_word_category = PWC[ W[word] ][cid]

                        # apply Pr(word|category)
                        if M == 'multivariate':
                                if word_count == 0:
                                        pr_word_category = \
                                         1.0 - pr_word_category
                                pr_category_words += math.log(pr_word_category)
                        elif M == 'multinomial':
                                pr_category_words += word_count * \
                                 math.log(pr_word_category)
                prs.append( [cid,pr_category_words] )

        # find the max Pr(category|words)
        max_cid = prs[0][0]
        max_pr = prs[0][1]
        for cid,pr in prs:
                if pr > max_pr:
                        max_cid = cid
                        max_pr = pr

        # return the category of ID max_cid
        return max_cid

def test(data):
        '''
        Purpose
                Return True if the document is accurately categorized and False
                 otherwise.
        Precondition
                data is a six-tuple containing document path, PC, PWC, C, W,
                 and M.
        '''
        # unpack data
        document_path,PC,PWC,C,W,M = data

        # extract expected ID
        category = document_path.split('/')[-2]
        expected_id = C[category]

        # extract actual ID
        actual_id = categorize(document_path,PC,PWC,C,W,M)

        # return test result
        return expected_id == actual_id

if __name__ == '__main__':
```

```
# check command line arguments
if len(sys.argv) != 7:
        print 'Usage: python categorize.py ' + \
         '<database> <user> <password> ' + \
         '<model> <test_dir> <num_features>'
        sys.exit(-1)

# connect to database
try:
        db = util.DB(sys.argv[1],sys.argv[2],sys.argv[3])
except MySQLdb.Error,e:
        print 'Error %d: %s' % (e.args[0],e.args[1])
        sys.exit(-1)

# get event model
M = sys.argv[4]
if M not in ['multivariate','multinomial']:
        print "Error: <model> must be either " + \
         "'multivariate' or 'multinomial'"
        sys.exit(-1)

# prepare data for document categorization
PC = db.get_category_probabilities()
PWC = db.get_word_category_probabilities()
C = db.get_category_dict()
W = db.get_word_dict( int(sys.argv[6]) )

# disconnect from database
try:
        db.close()
except:
        print 'Error %d: %s' % (e.args[0],e.args[1])
        sys.exit(-1)

# mark start time
start_time = time.time()

# get correctly-categorized and total document counts
arg_list = []
for document_path in glob.glob(sys.argv[5]+'/*/*'):
        data = (document_path,PC,PWC,C,W,M)
        arg_list.append(data)
num_correct = 0
num_total = 0
pool = Pool(processes=8)
for is_correct in pool.map(test,arg_list):
        if is_correct:
                num_correct += 1
        num_total += 1

# mark end time
end_time = time.time()
```

```
# print categorization summary
print '<experiment>'
print '\t<num_features>'+sys.argv[6]+'</num_features>'
print '\t<accuracy>%.3f</accuracy>' % \
 (float(num_correct)/float(num_total))
print '\t<execution_time>%.3f</execution_time>' % (end_time-start_time)
print '</experiment>'
```

Bibliography

[1] A. M. de Jesus Cardoso Cachopo (Mestre), *Improving Methods for Single-label Text Categorization*. PhD thesis, Technical University of Lisbon, 2007.

[2] A. McCallum and K. Nigam, "A comparison of event models for naive bayes text classification," in *IN AAAI-98 WORKSHOP ON LEARNING FOR TEXT CATEGORIZATION*, pp. 41–48, AAAI Press, 1998.

[3] "Extensible markup language (xml)." http://www.w3.org/XML/.

[4] "gnuplot homepage." www.gnuplot.info/.

[5] "Reuters-21578." http://www.daviddlewis.com/resources/testcollections/reuters21578/.

[6] "20 newsgroups." http://qwone.com/ jason/20Newsgroups/.

[7] "The 4 universities data set." http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-20/www/data/.

[8] "Datasets for single-label text categorization." http://web.ist.utl.pt/ãcardoso/datasets/.

[9] "The 2012 acm computing classification system." http://www.acm.org/about/class/2012.

[10] "Acm digital library." http://dl.acm.org/.

[11] L. S. Larkey and W. B. Croft, "Combining classifiers in text categorization," in *Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '96, (New York, NY, USA), pp. 289–297, ACM, 1996.

[12] G. Mecca, S. Raunich, and A. Pappalardo, "A new algorithm for clustering search results," *Data & Knowledge Engineering*, vol. 62, pp. 504–522, Sept. 2007.

[13] D. G. Roussinov and H. Chen, "Information navigation on the web by clustering and summarizing query results," *Information Processing & Management*, vol. 37, pp. 789–816, Oct. 2001.

[14] Z. Wei, H. Zhang, Z. Zhang, W. Li, and D. Miao, "A naive bayesian multi-label classification algorithm with application to visualize text search results," *International Journal of Advanced Intelligence*, vol. 3, pp. 173–188, July 2011.

[15] O. Zamir and O. Etzioni, "Grouper: a dynamic clustering interface to web search results," *Computer Networks*, vol. 31, pp. 1361–1374, May 1999.

[16] F. Peng, D. Schuurmans, and S. Wang, "Augmenting naive bayes classifiers with statistical language models," *Information Retrieval*, vol. 7, pp. 317–345, Sept. 2004.

[17] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz, "A bayesian approach to filtering junk e-mail," in *AAAI Workshop on Learning for Text Categorization*, 1998.

[18] S. Eyheramendy, D. Lewis, and D. Madigan, "On the naive bayes model for text categorization," in *Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics*, 2003.

[19] D. D. Lewis, "Naive (bayes) at forty: The independence assumption in information retrieval," in *Proceedings of the 10th European Conference on Machine Learning*, ECML '98, (London, UK, UK), pp. 4–15, Springer-Verlag, 1998.

[20] T. Joachims, "Text categorization with suport vector machines: Learning with many relevant features," in *Proceedings of the 10th European Conference on Machine Learning*, ECML '98, (London, UK, UK), pp. 137–142, Springer-Verlag, 1998.

[21] N. Friedman and M. Goldszmidt, "Building classifiers using bayesian networks," in *Proceedings of the thirteenth national conference on Artificial intelligence - Volume 2*, AAAI'96, pp. 1277–1284, AAAI Press, 1996.

[22] M. Sahami, "Learning limited dependence bayesian classifiers," in *In KDD-96: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pp. 335–338, AAAI Press, 1996.

[23] D. Koller and M. Sahami, "Hierarchically classifying documents using very few words," in *Proceedings of the Fourteenth International Conference on Machine Learning*, ICML '97, (San Francisco, CA, USA), pp. 170–178, Morgan Kaufmann Publishers Inc., 1997.

[24] A. K. McCallum, "Multi-label text classification with a mixture model trained by em," in *AAAI 99 Workshop on Text Learning*, 1999.

[25] D. D. Lewis and M. Ringuette, "A comparison of two learning algorithms for text categorization," in *In Third Annual Symposium on Document Analysis and Information Retrieval*, pp. 81–93, 1994.

[26] D. L. Olson and D. Delen, *Advanced Data Mining Techniques*. Springer, 2008.