What's the Code?
Automatic Classification of Source Code Archives

Secil Ugurel[1], Robert Krovetz[2], C. Lee Giles[1,2,3], David M. Pennock[2],
Eric J. Glover[2], Hongyuan Zha[1]

[1]Department of Computer Science and Engineering
The Pennsylvania State University
220 Pond Lab., University Park, PA 16802
{ugurel, zha} @cse.psu.edu

[2]NEC Research Institute
4 Independence Way, Princeton, NJ 08540
{krovetz, dpennock, compuman}
@research.nj.nec.com

[3]School of Information Sciences and Technology
The Pennsylvania State University
001 Thomas Bldg, University Park, PA, 16802
giles@ist.psu.edu

ABSTRACT
There are various source code archives on the World Wide Web. These archives are usually organized by application categories and programming languages. However, manually organizing source code repositories is not a trivial task since they grow rapidly and are very large (on the order of terabytes). We demonstrate machine learning methods for automatic classification of archived source code into eleven application topics and ten programming languages. For topical classification, we concentrate on C and C++ programs from the Ibiblio and the Sourceforge archives. Support vector machine (SVM) classifiers are trained on examples of a given programming language or programs in a specified category. We show that source code can be accurately and automatically classified into topical categories and can be identified to be in a specific programming language class.

1. INTRODUCTION
Software reuse is the process of creating software systems from existing software rather than building software systems from scratch. Software reuse is an old idea but for various reasons has not become a standard practice in software engineering [14] even though software reuse should increase a programmer's productivity by reducing the time spent on developing similar codes. Seemingly, programmers benefit from a repository where pre-written code is archived since such archives reportedly have over 100,000 users. However, software reuse does not only mean use of existing code [5]; it also involves organization and use of conceptual information. Thus, there should be methods so that programmers locate useful existing code quickly and easily.

SIGKDD '02, July 23-26, 2002, Edmonton, Alberta, Canada.

There are various source code archives and open source sites on the World Wide Web. If the programs in such sites are correctly classified and topically components are useful classifications, software reuse would be greatly facilitated. But how are the software programs categorized? In most archives programs are classified according to programming language and application topic. A programmer attempting to organize a collection of programs would most likely categorize resources based on the source code itself, some design specifications and the documentation provided with the program. But to understand which application category the code belongs to, it is very likely the programmer would try to gather natural language resources such as comments and README files rather than the explicit representation of the algorithm itself. Information in natural language can be extracted from either external documentation such as manuals and specifications or from internal documentation such as comments, function names and variable names. This seems reasonable since algorithms do not clearly reflect human concepts but comments and identifiers do [8]. But to identify the programming language, the programmer or administrator will look at the code itself and distinguish some of the keywords without trying to understand the algorithm. This should be straightforward since almost every language has its own reserved keywords and syntax. However, archives can be large and are rapidly changing, which makes manual categorization of software both costly and time consuming. If our goal is automatic categorization, then we believe it is a good idea to take advantage not only of the natural language information available in documentation but also the code itself.

Researchers have applied different learning techniques for text categorization: bayesian models, nearest neighbor classifiers, decision trees, support vector machines (SVMs) and neural networks. In text classification, each document in a set is represented as a vector of words. New documents are assigned to predefined categories using textual content. Recently, SVMs have been shown to yield promising results for text categorization [6, 7, 11]. Although programming languages are written in a manner different from natural languages and have some commented information, programming languages have specific keywords and

features that can be identified. Using these characteristics we show that text categorization techniques can also be effective for source code classification.

To build a classifier our first and maybe most important step is the extraction of features. For programming language classification, our feature set consists of 'tokens' in the source code and/or words in the comments. For the topical classification, we generate our features from words, bigrams, and lexical phrases extracted from comments and README files, and header file names extracted from the source code. We perform feature selection using the expected entropy loss. Each program is then represented as a binary feature vector. For each specific class, we use these vectors to train an SVM classifier. In order to evaluate the effectiveness of our approach, we measure the true-positive and false-positive rates for each class and the overall accuracy.

The rest of the paper is organized as follows. Section 2 summarizes background information and related work. In Section 3 we introduce our data set. In section 4 we describe our methodology and algorithm. Results are presented in Section 5 and conclusions are in Section 6.

2. BACKGROUND

2.1 Related Work

Various software identification and reuse problems explained in the literature vary in terms of the techniques they use and the features of programs they take advantage of.

Rosson and Carroll [18] examined the reuse of programs for the Smalltalk language and environment. They presented empirical results of the reuse of user interface classes by expert Smalltalk programmers. They observed extensive reuse by users and that the programmers searched implicit specifications for reuse of the target class and evaluated the contextualized information repeatedly. The programmers used and adapted code when the information provided matched their goals. Etzkorn and Davis [8] designed a system called *Patricia*, that automatically identified object-oriented software components through understanding comments and identifiers. They found object-oriented code more reusable than functionally-oriented code. Patricia uses a heuristic method deriving information from linguistic aspects of comments and identifiers and from other non-linguistic aspects of object-oriented code such as a class hierarchy. Merkl [16] suggested organizing a library of reusable software components by using self-organizing neural networks. Their approach is based on clustering the software components into groups of semantically similar components. They use keywords automatically extracted from the manual of software components. Each component is represented by this set of keywords, which does not include stop word lists. These representations are utilized to build the keywords-components matrix or the vector space model of the data. Each column of the matrix, which corresponds to the software components, is used to train the neural network. Search tools for source code are also important for software reuse. Chen et.al. [4] build a tool called CVSSearch that uses fragments of source code using Concurrent Version Systems (CVS) comments and makes use of the fact that CVS comments describe the lines of code involved. Evaluations of their technique show that CVS comments provide valuable information that complements content based matching. Henninger [10] also studied software components. Their approach investigates the use of a retrieval tool called CodeFinder, which supports the process of retrieving software components when information needs are not well defined and the users are not familiar with vocabulary used in the repository.

2.2 Expected Entropy Loss

How do we decide which features to select? Expected entropy loss is a statistical measure [1] that has recently been successfully applied to the problem of feature selection for information retrieval [8]. Expected entropy loss is computed separately for each feature. It ranks the features lower that are common in both the positive set and the negative set but ranks the features higher that are effective discriminators for a class. Glover et. al. [9] used this method for feature selection before training a binary classifier. We use the same technique. Feature selection increases both effectiveness and efficiency since it removes non-informative terms according to corpus statistics [19]. A brief description of the theory [1] is as follows.

Let C be the event that indicates whether a program is a member of the specified class and let f be the event that the program contains the specified feature. Let $\overline{C}$ and $\bar{f}$ be their negations and Pr( ) their probability.

The prior entropy of the class distribution is

$$e \equiv -\Pr(C)\lg\Pr(C) - \Pr(\overline{C})\lg\Pr(\overline{C})$$

The posterior entropy of the class when feature is present is

$$e_f \equiv -\Pr(C|f)\lg\Pr(C|f) - \Pr(\overline{C}|f)\lg\Pr(\overline{C}|f)$$

likewise, the posterior entropy of the class when the feature is absent is

$$e_{\bar{f}} \equiv -\Pr(C|\bar{f})\lg\Pr(C|\bar{f}) - \Pr(\overline{C}|\bar{f})\lg\Pr(\overline{C}|\bar{f})$$

Thus the expected posterior entropy is

$$e_f \Pr(f) + e_{\bar{f}} \Pr(\bar{f})$$

and the expected entropy loss is

$$e - e_f \Pr(f) + e_{\bar{f}} \Pr(\bar{f})$$

Expected entropy loss is always nonnegative, and higher scores indicate more discriminatory features.

2.3 Support Vector Machines

We use support vector machines (SVMs) for the binary classification task. Due to space restrictions, please see [2,3] for more details on SVMs. Support vector machines are generally applicable for text categorization problems and outperform other methods [11]. Their ability to handle high dimension feature vectors, to reduce problems caused by over-fitting, and to produce solutions robust to noise makes them a well-suited approach to text classification [15]. We choose a SVM classifier because our problem is similar to that of text classification.

3. DATA AND FEATURE SETS

We gathered our sample source code files and projects from different archives on the Internet including the Ibiblio Linux

Archive[1], Sourceforge[2], Planet Source Code[3], Freecode[4] and from pages on the web that include code snippets.

Ibiblio archives over 55 gigabytes of Linux programs and documentation freely available for download via FTP and/or WWW access. The Ibiblio archive includes binary files, images, sound and documentation as well as the source files of programs. It also archives different versions of the same project. The primary programming languages of the projects in the archive are C and C++. The projects are not classified into programming languages. The Ibiblio[1] archive is organized hierarchically into the following categories: applications, commercial, development tools, games, hardware and drivers, science and system tools. SourceForge.net[2] is owned by Open Source Development Network, Inc. and claims to be the world's largest open source development website. At the time of our study Sourceforge hosted 33,288 projects and had 346,328 registered users. The Sourceforge archive is categorized under 43 different programming languages and 19 topics. Programming languages include both popular languages such as C/C++, Java, Fortran and Perl, and less popular ones such as Logo and Zope. Users also have the capability of browsing Sourceforge projects by development status, environment, intended audience, license, natural language and operating system. Planet Source Code[3] claims to be the largest public programmer database on the web. During our analysis it had 4,467,180 lines of code. The code files are not organized by application area. They have 9 programming language categories. Free Code[4] is also owned by Open Source Development Network. The Free Code archive is organized under 10 topical categories and includes the programming language information for each project.

To train the topic classifier we downloaded examples from the Ibiblio and the Sourceforge archives, and concentrated on C/C++. For programming language classification we downloaded files from all of the resources. For our analysis, we select ten popular programming languages: ASP, C/C++, Fortran, Java, Lisp, Matlab, Pascal, Perl, Python and Prolog. These popular languages are used in a wide range of applications. For each class we randomly grouped our samples into disjoint training and testing sets. The training data consists of 100 source code files and the test data consists of 30 source code files from each category. Our experiments include those with comments included in the files and without comments. Although comments are used for giving contextual information about the program, we speculate that they might also be helpful in finding out the programming language of the code. As far as the topics are concerned, we selected categories/subcategories that contain sufficient number of projects in both resources (Ibiblio and Sourceforge) and eliminated the ones with a few projects or with no source code files in them. Another reason that we chose a different categorization was to evaluate the chance of mis-classification, which makes the task more difficult. Thus, we have category pairs that are well separated (e.g., "database" and "circuits") as well as category pairs that are quite similar (e.g., "games" and "graphics"). Table 1 lists our categories and the number of software programs we used for each from 2 resources.

Table 1. Number of programs used from each topic

CATEGORY	IBIBLIO	SOURCEFORGE
CIRCUITS	30	25
DATABASE	31	33
DEVELOPMENT	75	30
GAMES	209	31
GRAPHICS	190	36
MATHEMATICS	30	30
NETWORK	270	30
SERIAL COMMUNICATION	40	30
SOUND	222	31
UTILITIES	245	31
WORD PROCESSORS	11	24

4. METHODOLOGY

Our system consists of three main components; the feature extractor, vectorizer and the SVM classifier. There are also four supplementary modules, which are necessary for topical classification of programs: the text extractor, filter, phrase extractor and the stemmer. Our system works in two distinct phases: training and testing. Each of the components and phases are explained in the sections below.

4.1 Feature Extractor

Examples in each category are considered as the positives and the rest of the examples are counted as negatives. We compute the probabilities for the expected entropy loss of each feature as follows:

$$\Pr(C) = \frac{numberOfPositiveExamples}{numberOfExamples}$$

$$\Pr(\bar{C}) = 1 - \Pr(C)$$

$$\Pr(f) = \frac{numberOfExamplesWithFeatureF}{numberOfExamples}$$

$$\Pr(\bar{f}) = 1 - \Pr(f)$$

$$\Pr(C|f) = \frac{numberOfPositiveExamplesWithFeatureF}{numberOfExamplesWithFeatureF}$$

$$\Pr(\bar{C}|f) = 1 - \Pr(C|f)$$

$$\Pr(C|\bar{f}) = \frac{numberOfPositiveExamplesWithoutFeatureF}{numberOfExamplesWithoutFeatureF}$$

$$\Pr(\bar{C}|\bar{f}) = 1 - \Pr(C|\bar{f})$$

The feature extractor indexes each file and computes the expected entropy loss for each feature. Then the features are sorted by descending expected entropy loss. Some features that appear more frequently in the negative set might also have large expected entropy loss values. We call these features "negative features". Thus, a feature that has a higher frequency in the negative set can also be distinguishing for a category. Our feature extractor does not eliminate stop words and can take bigrams into account. It can also consider lexical phrases as features using the output of the phrase extractor module. By default, the feature extractor module

[1] www.ibiblio.org/pub/linux
[2] www.sourceforge.net
[3] www.planetsourcecode.com
[4] www.freecode.com

removes the features that only appear in a single file. It is also capable of eliminating features that occur below a given threshold of positive and negative examples.

4.1.1 Programming Language Feature Extractor

Features correspond to tokens in the code and words in the comments. We define a "token" to be any alphabetical sequence of characters separated by non-alphabetical characters. We do not consider any numeric values or operators as tokens. Tokens are gathered by splitting the source code text at any non-alphabetical character such as white space. For example, in an expression like "if (topicClass10 = 'network')", the tokens will be "if", "topicClass" and "network".

The top 20 features selected by expected entropy loss from each of 10 classes were used to generate a set of 200 features. We excluded features that occurred in less than 10% of both the positive and negative set in the vocabulary.

4.1.2 Application Topic Feature Extractor

The feature extractor for topic classification uses 3 different resources: comments, README files and the code. The extraction of comments from the source code files and identification of README files for each program is performed by the text extractor module. We do not run the feature extractor directly on the output of the text extractor but preprocess the data by filtering, stemming and phrase extraction modules. The filter module is written to eliminate data that are uninformative for identification of an application category, such as license and author information, which appear in any program. Although the expected entropy loss technique is likely to rank these features very low, filtering decreases the size of our files and improves the speed of our algorithms.

We used KSTEM for grouping morphological variants. KSTEM uses lexicon and morphological rules to determine which word forms should be grouped together. It is designed to avoid grouping word forms that are not related in meaning [13]. Stemming is useful because, for example, "colors" and "color", which typically refer to the same concept, are merged into one feature. Lexical phrases are important because they reduce ambiguity (they are usually less ambiguous than the component words in isolation). In addition, sometimes terms are only meaningful as phrases (i.e. "work station"). That is, sometimes a phrase is essentially a word with an embedded space.

We combine the README files and the comments for each program separately and pull out single words, bigrams and lexical phrases. From the code itself we just include the header file names. The top 100 features from each 11 categories are combined to generate a set of 1100 features. We believe 100 features would be sufficient for a good classification. We have a selection threshold of 7.5% for application topic classification.

4.2 Vectorizer

The vectorizer generates feature vectors for each program/source code file. We do not consider the frequency of the features. The elements of the vectors consist of 1s and 0s. A '1' in a feature vector means that the corresponding feature exists in the corresponding example and a '0' means that it does not. The vectorizer module uses the features extracted from the training data for creating vectors for both the test and the training set.

4.3 SVM Classifier

The SVM classifier is trained on vectors generated from the training set in which each document has a class label. It returns the overall accuracy of the classification, which is the percentage of programs that are categorized correctly.

We use "LIBSVM – A Library for Support Vector Machines" by Chang and Lin [3], which is integrated software for support vector classification, regression and distribution estimation. LIBSVM is capable of performing cross validation, multi-categorization and using different penalty parameters in the SVM formulation for unbalanced data. LIBSVM uses the "one-against-one" approach [12] for multi-class classification. In the one-against-one approach, k(k-1)/2 classifiers are constructed where k is the number of classes. Each classifier trains data from two different classes. Chang and Lin [3] utilize a voting strategy where each example is voted against a class or not in each binary classification. At the end the program is assigned to the class with the maximum number of votes. In the case that two classes have the same number of votes, the one with the smaller index is selected. There is also another approach for multi-class classification called "one-against-all". In this technique, the number of SVM models as many as the number of classes are constructed. For each class, the SVM is trained with all the examples in that class as positives and the rest of the examples as negatives. Previous research has shown that one-against-one approach for multi-class categorization outperforms the one-against-all approach[17].

5. EXPERIMENTAL RESULTS

5.1 Programming Language Classification

For programming language classification, we performed our experiments both with the comments included in the code and without comments to ascertain the impact of comments. The feature extraction step gives us a list of words that best describes a programming language class.

Table 2. Top 10 features for each class

CLASS	COMMENTS ARE INCLUDED	COMMENTS ARE EXCLUDED
ASP	asp, dim, vbscript, td, head	asp, vbscript, dim, td, language
C\C++	struct, void, sizeof, include, unsigned	struct, void, ifdef, sizeof, include
FORTRAN	subroutine, pgslib, logical, implicit, dimension	subroutine, logical, pgslib, dimension, implicit
JAVA	throws, jboss, java, ejb, lgpl	jboss, throws, java, package, util
LISP	defun, lisp, setq, emacs,	defun, let, setq, progn,
MATLAB	zeros, -type, denmark, veterinary, -license	zeros, -name, -type, -string, plot
PASCAL	unit, sysutils, procedure, synedit, mpl	implementation, unit, luses, procedure, sysutils
PERL	speak, voice, my, said, print	my, speak, voice, said, print
PYTHON	def, moinmoin, py, jhermann, hermann	def, moinmoin, py, copying, rgen
PROLOG	prolog, predicates, diaz, descr, fail	-if, fail, built, bip, atom

Table 2 lists only the top 5 words when comments are used with the code and when comments are filtered. A minus sign indicates

the negative features (ones that are more frequent in the negative set compared to the positive set). We generated a set of 200 features by taking the top 20 features from each class. Our training data consists of 100 source code files and test data consists of 30 source code files from each language class.

Table 3 lists the true-positive rates and false-positive rates for each language class and the overall accuracy of our classifier.

Table 3. TP rate, FP rate and the accuracy of the classifier

	COMMENTS ARE INCLUDED		COMMENTS ARE EXCLUDED	
CLASS	TP RATE	FP RATE	TP RATE	FP RATE
ASP	100.00%	0.34%	90.00%	1.75%
C\C++	93.33%	0.00%	93.33%	0.00%
FORTRAN	81.48%	0.68%	95.24%	0.35%
JAVA	70.00%	0.00%	63.33%	0.00%
LISP	93.33%	0.00%	83.33%	0.00%
MATLAB	96.30%	7.53%	100.00%	8.39%
PASCAL	86.21%	0.00%	86.66%	0.00%
PERL	100.00%	1.03%	93.33%	1.05%
PHYTON	96.66%	0.00%	89.65%	1.05%
PROLOG	72.41%	1.37%	82.14%	0.00%
ACCURACY	89.041%		87.41%	

We think that the performance of each class highly depends on the programming language that is being classified and the overlap between the tokens in source code files. For this reason, we explored the intersections between the top 100 features of each class and presented the results in Table 4.

Table 4. Overlap of features between categories. The upper triangle shows the overlap rates when comments are included. The lower triangle shows when comments are excluded.

CAT	ASP	C\C++	FORT	JAVA	LISP	MATL	PAS	PERL	PHYT	PRO
ASP	-	1%	1%	1%	0%	3%	0%	1%	0%	1%
C\C	1%	-	1%	4%	1%	0%	1%	0%	0%	20%
FOR	2%	2%	-	4%	0%	22%	0%	23%	1%	5%
JAVA	0%	7%	0%	-	1%	3%	1%	5%	2%	1%
LISP	0%	1%	1%	0%	-	0%	1%	1%	0%	1%
MATL	1%	2%	18%	6%	0%	-	0%	36%	1%	7%
PAS	1%	1%	2%	1%	1%	1%	-	0%	2%	1%
PERL	1%	2%	9%	6%	2%	23%	0%	-	2%	6%
PHYT	0%	2%	3%	5%	1%	5%	0%	6%	-	0%
PRO	1%	1%	13%	3%	0%	18%	1%	13%	2%	-

The upper triangle of the table lists the overlaps when features are extracted from both the code and the comments. The lower triangle, on the other hand, lists the overlaps when features are gathered only form the code. We observe that in both cases, Matlab class has the highest overlap percentages with other language classes (especially with Perl) and it is also the class with the highest false positive as well. It is also true that most of the examples that are not correctly classified are assigned to the Matlab class. On the other hand the effect of the use of comments in programming language classification depends on the language. Although comments help to increase the overall accuracy of classification, they have a bad effect on identification of Fortran, Matlab, Pascal and Prolog classes.

5.2 Application Topic Classification

To test our method for topical classification we performed five different experiments on different data sets using combinations of the three types of features: single words, lexical phrases and bigrams. In each experiment, we chose 100 features from each of 11 categories and generated a set of 1100 features. Features were selected according to their expected entropy loss. Table 5 lists the abbreviations of experiments and the feature types used in each experiment. These sets were generated from both the Sourceforge and the Ibiblio archive.

Table 5. Types of features used in each experiment

EXP.	TOP TEN FEATURES EXTRACTED
SW	Top 100 features from single words
LP	Top 100 features from lexical phrases
2G	Top 100 features from bigrams
SW2G	Top 100 features from single words and bigrams
SWLP	Top 100 features from single words and lexical phrases

The outputs of the feature extractor were promising for each category. We were able to select the features, from which one can easily guess the corresponding category. For example, Table 6 tabulates the top five words and lexical phrases extracted from the Ibiblio Archive. It is not surprising that we have "calculator" for the mathematics class, "high score" for the games class and "database" for the database class. On the other hand, some of the features are shared among the categories since they have multiple meanings; for example, "play" appears in both the sound and the games classes. Another observation is that the utilities category has more negative features than positive ones. This means that the words like "play" and "client" are unlikely to appear in the utility programs and the "socket.h" library is not included in most of them.

To evaluate our classifier and to be able to find the appropriate penalties for training, we first applied 5-fold cross validation to each data set (Sorceforge and Ibiblio) separately and to the combined sets. In 5-fold cross validation, the data is divided into 5 subsets and each time one subset is used as the test set and the 4 subsets are used for training. We did not use the same archive for both training and testing because the number of examples in some of the categories in an archive were not sufficient. Another factor about our data is that it is unbalanced. For example the number of programs in the word processors category is 11 where it is 270 in the network category. Thus, we used the weighted version of the SVM and changed the penalty parameters (C) for categories. Penalty for each category is computed by multiplying the weights by the specified cost C. We chose the linear kernel function and assigned 100 to C.

Table 7 lists the accuracies of the cross validations performed on the Sourceforge, the Ibiblio archive and the on the combined sets for each experiment. We have 2 experiments for the combined sets because one set uses the features extracted from the Ibiblio Linux and the other uses the features extracted from Sourceforge.

Table 6. Top ten words and lexical phrases from each category of Ibiblio archive

CLASS	TOP FIVE WORDS	TOP FIVE LEXICAL PHRASES
CIRCUITS	circuit, spice, pin, simulator, transistor	standard cell, transfer curve, circuit interface, cell library, short channel
DATABASE	sql, database, query, postgresql, libpq	database, database system, database server, sql statement, method code
DEVELOPMENT	class, thread.h, new.h, iostream.h, malloc	class library, first item, class hierarchy, global function, header file
GAMES	game, games, play, score, xlib.h	high score, new game, new level, computer player, map
GRAPHICS	image, jpeg, gif, ppm, pixel	image, independent jpeg, jpeg library, jpeg software, image file
MATH	calculator, mathematics, exponent, math, fractal,	plot function, radix mode, real numbers, palette change, complex numbers
NETWORK	socket.h, netdb.h, in.h, ip, inet.h	ip address, security fix, error output, backup copy, libc version
SERIAL COMM.	modem, zmodem, voice, fax, serial	serial port, modem device, script language, voice modem, incoming data
SOUND	soundcard.h, sound, audio, mixer, soundcard	sound driver, cd player, sound card, audio device, track
UTILITIES	-game, -netdb.h, -socket.h, -client, floppy	floppy disk, illegal value, block device, other locale, appropriate system
WORD PROCES.	tex, dvi, latex, lyxrc, tetex	latex command, style sheet, dvi driver, default value, vertical scale

Table 7. Cross validation accuracies. In the third data set features used are extracted from Sourceforge and in the fourth data set features used are extracted from the Ibiblio Archive.

DATA SET	ACCURACY				
	SW	LP	2G	SW2G	SWLP
SOURCEFORGE	43.20%	19.64%	27.79%	38.37%	41.39%
IBIBLIO	72.51%	49.96%	56.24%	72.36%	72.58%
COMBINED (SOURCEFORGE)	64.13%	33.73%	36.10%	56.77%	60.22%
COMBINED (IBIBLIO)	64.55%	46.50%	50.53%	67.34%	66.80%

Table 8. TP, FP rates and the overall accuracies for each experiment using the features from Ibiblio

CLASS	SW		2G		LP		SW2G		SWLP	
	TP %	FP %	TP %	FP %	TP %	FP %	TP %	FP %	TP %	FP %
CIRCUIT	18.51	2.08	28.57	14.02	21.43	2.46	28.57	1.60	28.57	1.11
DATAB.	60.60	1.11	38.71	3.21	19.35	2.96	45.16	1.11	75.16	0.99
DEVEL.	40.38	4.05	50.94	5.84	26.41	11.55	54.72	5.58	50.94	5.84
GAMES	71.66	4.58	69.17	4.02	60.83	6.24	80.83	3.88	80.00	4.02
GRAPH.	64.60	5.63	56.64	8.24	50.44	9.20	72.57	10.16	70.80	8.79
MATH	30.00	1.11	6.67	2.34	30.00	3.45	26.67	1.36	30.00	1.85
NET.	82.66	7.38	59.33	4.48	52.57	3.91	84.00	4.34	82.00	5.21
SERIAL.	31.43	1.48	11.43	1.61	25.71	6.45	42.86	0.87	34.28	0.50
SOUND	77.95	2.66	65.08	4.33	49.21	5.03	82.54	2.24	83.33	2.94
UTIL.	67.15	10.94	34.53	4.13	40.29	7.55	52.52	6.84	54.68	6.84
WORD P.	5.55	0.12	23.53	2.55	29.41	2.79	17.65	0.36	23.53	0.97
ACCUR.	64.25%		50.24%		44.65%		66.39%		65.80%	

When we compare the two data sets, Ibiblio performs better than Sourceforge. Although we apply cross validation, the reason for the poor performance appears to be the number of examples in the Sourceforge data. For most of the categories, we used fewer examples from the Sourceforge than the Ibilio archive. As far as the types of features, single words together with lexical phrases are the most helpful feature group for classification. Although, lexical phrases alone do not perform well, they increase the accuracy of the cross validation on Ibiblio archive when used with the single words. Single words with bigrams are also useful and outperform the other techniques for the last data set.

Second, we split our combined data set to two subsets and used one subset for training and the other for testing. We used the features extracted from the Linux archive in this experiment. Table 8 shows the true positive and false positive rates and the overall accuracy of the SVM classifier trained by the features from the Ibiblio Archive and tested on the combined set. Similar to the programming language classification, single words when used with bigrams and lexical phrases perform the best on overall. This is also true for each category but the utilities. Between the categories, the database, games, graphics, network and the sound classes performed much better than the other classes. This is again related to the few examples we have in the other classes and the fuzziness of the utilities class. We observe that the utilities class always has a high false positive rate.

Table 9. TP, FP rates and the overall accuracies for each experiment using the features from Sourceforge

CLASS	SW		2G		LP		SW2G		SWLP	
	TP %	FP %	TP %	FP %	TP %	FP %	TP %	FP %	TP %	FP %
CIRCUIT	17.86	1.48	25.00	12.80	21.43	3.81	32.14	2.95	25.00	1.84
DATAB.	41.93	0.86	19.35	1.48	16.13	2.83	51.61	1.48	38.71	0.99
DEVEL.	50.94	5.46	26.41	6.98	20.75	6.34	54.71	1.05	37.74	4.95
GAMES	74.17	4.30	39.17	1.32	37.50	12.62	77.50	4.58	71.67	6.10
GRAPH.	82.30	14.29	35.40	8.24	38.05	12.91	70.80	15.80	73.45	16.08
MATH	43.33	1.48	36.67	3.33	23.33	1.48	36.67	1.36	43.33	1.23
NET.	80.00	2.60	22.67	1.01	22.00	1.30	46.67	1.44	68.67	2.75
SERIAL.	22.86	1.61	20.00	5.71	22.86	5.46	25.71	1.61	25.71	1.36
SOUND	86.51	4.61	47.62	1.97	57.14	25.59	73.02	7.69	75.40	7.41
UTIL.	38.13	3.70	16.55	2.99	12.95	4.42	23.02	3.28	40.29	4.70
WORD P.	17.65	1.21	23.53	2.55	23.53	2.67	29.41	2.06	0.00	1.09
ACCUR.	64.60%		30.05%		30.88%		52.97%		57.48%	

In the third step, the classifier was trained with features from the Sourceforge archive and tested on the combined data set. Table 9 shows the accuracy of the SVM classifier for each category. This time experiments on single words have the highest performance. The classes that perform the best do not change for this experiment but the false positive rate of the graphics category is

worse than the others. When the two data sets are compared, not surprisingly the accuracy is higher when we train our classifier with features from the Ibiblio archive. Please note that we used the same penalties in each method of the experiments to be able to compare the feature types. However, weights can be different for each feature type to increase the overall accuracy.

6. CONCLUSIONS AND FUTURE WORK

Our experiments show that source code can be accurately classified with respect to programming language and application category. However the accuracy of this classification depends on many factors. The variance of our data, the application categories, the selection of features used, the information retrieval techniques and the programming language all affect performance.

We demonstrate an SVM based approach to programming language and topic classification of software programs. We train our classifier with automatically extracted features from the code, comments and the README files. For programming language classification, these features are tokens in the code and words in the comments. For topical classification, we use words, bigrams and lexical phrases in the comments and README files and header file names in the code as features. We perform feature selection by expected entropy loss values and train SVM classifiers using these features. Though our work shows promising results, there is much to explore, including the choice and number of feature vectors. Using values such as term frequency in the vectors, instead of binaries can improve the performance of our classifier. Our work for programming language classification can also be extended by adding more syntactic features together with the words. We believe that other properties of programming languages, such as the way comments are included and the tokens used for arithmetic or logical operations, will help in identifying the programming language.

These results imply that large archive collections of mixed data such as text and source code can effectively be automatically classified and categorized. We feel this will lead to more effective use of code archives and a reduction in duplication of programmer effort.

7. ACKNOWLEDGEMENTS

We gratefully acknowledge Gary Flake, Eren Manavoglu and Burak Onat for their comments and contributions.

8. REFERENCES

[1] Abramson, N. "Information Theory and Coding." McGraw-Hill, New York, 1963.

[2] Bennett, K. P. and Campbell, C. "Support vector machines: Hype or Hallelujah." *ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD) Expolarations 2(2)*:1-13, 2000.

[3] Chang, C. and Lin, C. "LIBSVM: A library for support vector machines." Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[4] Chen, A., Lee Y. K., Yao A. Y., and Michail A. "Code search based on CVS comments: A preliminary evaluation," (Technical Report 0106). School of Computer Science and Eng., University of New South Wales, Australia, 2001.

[5] Creps, R. G., Simos, M. A., and Prieto-Diaz R. "The STARS conceptual framework for reuse processes, software technology for adaptable, reliable systems (STARS)" (Technical Report). DARPA, 1992.

[6] Dumais, S. T. "Using SVMs for text categorization." *IEEE Intelligent Systems Magazine*, Trends and Controversies, 13(4):21-23, 1998.

[7] Dumais, S. T., Platt J., Heckerman D., and Sahami M. "Inductive learning algorithms and representations for text categorization." *Proceedings of the ACM Conference on Information and Knowledge Management*, 148-155, 1998.

[8] Etzkorn, L. and Davis, C. G. "Automatically identifying reusable OO legacy code." *IEEE Computer*, 30(10): 66-71, 1997.

[9] Glover, E. J., Flake, G. W., Lawrence, S., Birmingham, W. P., Kruger, A., Giles, L. C., and Pennock, D. M. "Improving category specific web search by learning query modification." *IEEE Symposium on Applications and the Internet (SAINT 2001)*, 23-31. San Diego, CA, US: IEEE, 2001.

[10] Henninger, S. "Information access tools for software reuse." *Systems and Software*, 30(3): 231-247, 1995.

[11] Joachims T. "Text categorization with support vector machines." *Proceedings of the Tenth European Conference on Machine Learning*, 137-142, 1999.

[12] Knerr, S., Personnaz, L., and Dreyfus, G. "Single layer learning revisited: a stepwise procedure for building and training a neural network." *Neurocomputing: Algorithms, Architectures and Applications*. J. Fogelman (Ed.), Springer-Verlag, 1990.

[13] Krovetz, R. "Viewing Morphology as an Inference Process." *Artificial Intelligence, 20,* 277-294, 2000.

[14] Krueger, C. W. "Software resuse." *ACM Computing Surveys,* 24(2):131-183, 1992.

[15] Kwok J. T. "Automated text categorization using support vector machines." *Proc. of the International Conference on Neural Information Processing*, 347-351, 1999.

[16] Merkl, D. "Content-based software classification by self-organization." *Proc. of the IEEE International Conference on Neural Networks*, 1086-1091, 1995

[17] Platt, J.C., Cristianini, N., and Shawe-Taylor, J. "Large margin DAGs for multiclass classification." *Advances in Neural Information Processing Systems 12*, 547-553. MIT Press, 2000.

[18] Rosson, M.B. and Carroll, J.M. "The reuse of uses in Smalltalk Programming." *ACM Transactions on Computer-Human Interaction, 3(3)*, 219-253, 1996.

[19] Yang, Y. and Pederson, J. "A comparative study on feature selection in text categorization." *Proceedings of the Fourteenth International Conference on Machine Learning (ICML'97)*, 412-420, 1997.