Finding Optimal Hardware Configurations For C Code

John Clark, Ilana Segall
{jpclark, isegall}@stanford.edu

December 11, 2009

1 Introduction

To look at a piece of C code - especially an uncommented one - is often not very enlightening. Even small pieces of code with one or two functions will often appear to perform the same array initializations, loops over the data, and basic arithmetic functions, without clearly indicating what the overall purpose of the routine is. This can be frustrating to those using the code as part of a package who are forced to understand it as a "black box," without completely understanding its properties. More specifically, attempting to find the most optimal way to run the code becomes a very difficult problem if there are, for example, several different choices of harware to choose from, and the actions of the program are a mystery.

In this problem, we choose to attack a tractable subproblem that we hope will be helpful for deciding how to run this software for a user with several different hardware options. Specifically, we have observed that most single code files perform one of two functions: data manipulation and computation. We hope to be able to successfully classify C code files into these two main categories to allow a user who has not been informed of its functionality to quickly determine its core purpose, and use this information to enhance performance of the overall project.

We choose to look at the assembly representation of these files, whose simple structure and more limited vocabulary we believe contain the key clues to functionality in order to perform this classification. Generally, the conversion to assembly is done "behind the scenes," and only a small percentage of programmers actually feel comfortable reading any particular type of assembly and understanding what it does, so we believe that assembly is underutilized in understanding code's functionality. Thus, we also hope that classifying code in this manner will provide a strong argument for using assembly to extract features of code for use in future machine learning applications pertaining to C or other programming languages, and eliminate the need to look at assembly directly.

2 Data

The data for our learning algorithms consists of C code files from one of the two groups: (1) 35 data manipulation algorithms and (2) 145 computational algorithms. We hypothesize that a learning algorithm able to distinguish well between code files from these two canonical

groups would be also able to categorize code files with respect to their expected performance on a hardware platform. We acquired the code files used in this project from online C repositories and the CBLAS library. The data manipulation code consists of searching and sorting algorithms including algorithms such as breadth and depth first search, Dijkstra's algorithm, the Floyd-Warshall algorithm, mergesort, and bubblesort, many of which have several samples created to perform the algorithms over different types of data structures. The CBLAS library is a C interface to the BLAS routines, which contain many low-level linear algebra producures such as matrix-matix operations and operations on sparse vectors.

3 Methods

We chose to use the assembly instructions of each code file as the features for this project. For each of the code files we produce the corresponding assembly language file by compiling the code file with the command "gcc -S". This produces a .s file which is the code compiled but not assembled or linked. Each line of the assembly file has the format: "$\langle Command \rangle \langle arguments \rangle$". We chose to use only the assembly command, the first token of most lines, since they carry the most information about the operation carried out by each line.

Once each code file was compiled into assembly we wrote a simple parser to extract all of the commands and their frequency of appearance in each code file. The parser opens and reads each file storing in the vector "tokenlist" commands which have not been seen before. This gives us in "tokelist" a list of all commands that appear in all the files which can be used as our dictionary.

A beneficial aspect of using assembly commands as the features of our learning algorithm is that that they are extremely regular tokens in the code files. There is no preprocessing necessary: each command appears exactly the same way every time it occurs in the file. Also since the commands are non-ambiguous, unlike with natural language tokens, the meaning of each command is completely contained in its name. In contrast, a negative aspect of using assembly commands as the features of our learning algorithm is that there are not very many of them. Our token list taken from 180 algorithmically very different code files contained only 48 tokens. This is a relatively small number of features from which to gather information, though in our case we found this number to be sufficient.

Because we are performing a binary classification, we chose to compare results from clustering, Naive Bayes, and a support vector machine. We chose to use an unsupervised learning algorithm, k-means clustering, to see if the code examples divided themselves into two groups different than the ones we had originally imagined, which could provide insight into our categorizations.

With Naive Bayes, we chose to approach the problem two ways. We used the standard Naive Bayes model, using a feature vector with every assembly command represented as a 0 or 1, where 1 indicates a feature's presence in the code example. We then also used the multinomial event model to see if the code classified better as text - specifically, if the number of times a command appeared in code was more informative than its presence. For both methods, we used Laplace smoothing and a custom (add $1 * 10^{-8}$) smoothing to avoid 0/0 errors but reduce the probability of an event we have not yet seen - especially in the case of the multinomial event model, there are relatively few occurences of each command

in every document, and we want to preserve accurate multinomial probabilities as well as we can.

Finally, we implemented a support vector machine to test the linear separability of the data. We used a simplified version of the SMO algorithm, which converged quickly enough that we did not feel the need to include all the optimizations in the full algorithm.

All of the tests described above were performed in MATLAB.

4 Results

4.1 Clustering

Because clustering does not have an explicit error metric and it would be very difficult to visualize our data (as the feature vector is in 48 dimensions), we examined both how the code divided into the two clusters and the composition of each cluster. For k = 2, we found that, of the computational code, about 36% was in cluster 1, and 64% in cluster 2, while for data manipulation code, about 20% was in cluster 1 and 80% in cluster 2. Alternatively, this means that cluster 1 composed of 67% is computational code and 33% data manipulation code, while cluster 2 contained 47% computational code and 53% data manipulation code. Clearly, clustering was not an effective algorithm to divide the code samples into two groups.

4.2 Naive Bayes

In the tables below we show some of the estimated generalization errors calculated by cross validation of each Naive Bayes algorithm. We performed two types of cross validation, k-fold with k = 6 or 7, and leave one out cross validation. We ran these cross validations for all four of the below described versions of Naive Bayes, and ran each once with all of our files and again with a subset of the computational files (because we had more than four times as many computation files as data manipulation ones, we wanted to assure that the algorithms performed robustly if the amount of each was even). Figure 1 below displays the best of the results from this analysis.

Clearly, all algorithms performed extremely well in this analysis - several checks were performed on the implementation of the algorithm to ensure that such low error was indeed calculated correctly. Interestingly, very little difference was seen between the standard Naive Bayes and multinomial event model error, indicating that the presence of an instruction in the code was as important as the frequency of its occurence. We also see that the custom smoothing we performed did not create a better alternative than Laplace smoothing, though because all tests returned such excellent results, we would have to examine its performance on a different data set to conclude its effectiveness. Also, we found a slightly lower error when we used our full data set, although the difference was negligable.

Figure 2 below shows the top five tokens that strongly indicate classification into one of the two categories. These results agree with what we would expect from such algorithms, especially the presence of the streaming SIMD extensions in the computation category, which are used for parallelization of a large number of computations.

# Data Manip = 35 # Computation = 35	Multinomial Event model	Naive Bayes
Standard Laplace smoothing	LOOCV: 0.0 K-fold(k=7): 0.0	LOOCV: 0.0143 K-fold(k=7): 0.0
Custom smoothing (.00000001)	LOOCV: 0.0 K-fold(k=7): 0.0	LOOCV: 0.0143 K-fold(k=7): 0.0
# Data Manip = 35 # Computation =145	Multinomial Event model	Naive Bayes
Standard Laplace smoothing	LOOCV: 0.0056 K-fold(k=6): 0.0056	LOOCV: 0.0056 K-fold(k=6): 0.0056
Custom smoothing (.00000001)	LOOCV: 0.0056 K-fold(k=6): 0.0056	LOOCV: 0.0 K-fold(k=6): 0.0

Figure 1: Estimated Generalization error for the Naive Bayes classification models. These estimates are taken using both k-fold cross validation and leave one out cross validation (LOOCV).

Naive Bayes, LOOCV	
Data Manipulation	**Computational**
SHRL: Shift right	HLT: Enter halt state
JGE: Jump if greater than equal to	NEGL: Negation
ANDL: Bitwise and	MOVSS: Move scalar single (SSE)
JG: Jump if greater than	MOVSD: Move scalar double (SSE2)
SARL: Shift arithmetically right	MOVAPS: Move aligned single precision (SSE)
	SSE=Streaming SIMD Extensions

Figure 2: These are the top five assembly instructions which most strongly indicate that a code file is either a data manipulation type algorithm (left column) or a computational type algorithm (right column).

4.3 SVM

Using k-fold cross validation with k = 7 and an equal number of data manipulation and computational files on our SVM, we obtained an error of 0.0143, which is also excellent. This is an interesting result considering how poorly the data clustered, strongly indicating that supervised learning is necessary for this classification. This indicates that though a separating hyperplane does very well in separating the data, the functional margins of the training examples are not very large relative to their distances from each other.

5 Conclusion

Both Naive Bayes and support vector machines appear to be excellent algorithms for classifying our code, and performed much better than the unsupervised k-means clustering. The presence of such low error in both trials indicates that the feature set was extremely indicative of the classification we desired, despite the fact that it is difficult to ascertain what assembly commands will be needed by looking at the direct C code. Assembly commands appear to make very powerful features for code classification.

6 Future Work

The original purpose of this project was to be able to determine the optimal hardware configuration to optimize the runtime of an unidentified piece of code. Due to several technical issues, we were unable to get a hardware simulator, SimpleScalar, to import the code we were interested in using as our training set. We do believe, however, that our classification could be used to help make this determination (perhaps by using its designation as data manipulation or computation as a feature) in the future. Using SimpleScalar would be an excellent way to create the training set for such a project.

Though our feature set was sufficient for this problem, future work in this area should also look into ways of encoding the location in the code file of the assembly command as well as its frequency of occurrence. We feel that this would provide another piece of information which would shed even more light on the algorithm behind the code, especially if a more complicated problem, such as the one above, is being explored. Many other features are available from the assembly code, such as the number of registers used and the arguments of the instructions.

References

[1] CBLAS library. *http://www.gnu.org/software/gsl/manual/html*$_n$*ode/GSL − CBLAS − Library.html.*

[2] Happy Codings. *http://www.c.happycodings.com/.*