CSE 515
Predicting Code Efficiency Automatically on the Google Code Jam dataset

Daniel Graf (1350014) **Jijiang Yan (0837209)**

1 Introduction

Background

Google Code Jam (link) is one of the largest online coding competition with over 10,000 participants every year. The competition consists of several rounds. In each round, participants need to solve a few tasks, each of which will be tested on two sets of inputs, a small one and a large one. The small input set can often be solved using brute force algorithms. For the big test set a more sophisticated implementation is required.

During the contest the execution of the program is done on the participant's machine. The participant has four minutes for the small and eight minutes for large input set to download the input file, run his code and upload the output together with his code back to Google's server. This way Google can just compare the output file with their solution and does not need to run the participant's code in their data center. After the end of the contest all correct submissions are published. The owner of the website go-hero.net (link) has collected all of these submissions and allows for queries by task, programming language and nationality of the participants.

By learning the efficiency on the two inputs, we studied whether it is possible to automatically predict the runtime of a given implementation, by only looking at static high-level features of the code, so without actually executing it.

Motivation

In many courses and classes students are asked to implement a specific algorithm or solve a certain problem and to submit their solutions to an automatic grading tool. This is the case for many classes in algorithms and data structures in universities worldwide and online (like coursera.org).

Also very popular are programming contests like the ACM International Collegiate Programming Contest (ICPC) or the International Olympiad in Informatics (IOI) for high school students. There students need to solve algorithmic puzzles in a limited amount of time. Often partial score is given according to the asymptotic complexity of the students implementation.

A fast automatic evaluation tool could help both the organizers and participants of such contests, as well as instructors and students of such courses. By detecting programs that stand out of the ordinary, organizers can effectively detect the submissions that they need to double check manually. So such a classifier can help organizers of coding contests to detect outliers and new types of solutions. Especially in real-time contests (like the IOI) or on grading systems with many users (like coursera) it is very hard to stay on top of all the submissions that come in, so it is essential that the grading system supports the organizers here.

From the students perspective such a tool can be helpful either while debugging during the contest or when analyzing and comparing the own solution with others afterwards. Statements of the form 90% of the programs that are faster than yours are using std::map might be a very helpful starting point in order to improve the students learning process. So a participant that tries to write a fast program for a task might also benefit from our classifier, as it can suggest possible reasons why the student's code is slow.

Goals

We set the following goals for our project:

- collect and prepare all the submissions from several tasks
- extract static features
- train and evaluate Naive Bayes and logistic regression classifiers for single tasks
- train and evaluate multi-task logistic regression classifiers

Previous Work

Related work has been done to detect source code plagiarism and outsourcing, like in [1], and probabilistic graphical models have been applied to related topics, like static analysis of source code in [2]. A method for multi-task logistic regression is given in [3]. We refer to the literature survey for a more detailed overview of previous work.

2 Methods

Data Collection

By parsing the archives on go-hero.net (link) we downloaded all the submissions in C++ for the tasks of the qualification round and the round 1A of the code jam 2013. We focused on submissions in C++, as this is by far the most popular language in the contest. For instance, 11342 of all the 23115 submissions to the problem 'lawnmower' in the qualification round were written in C++.

To get the set of programs that solve only the small input and the set of efficient programs that also solve large input we split the submissions as follows: If a contestant was successful on both inputs, we only considered his solution to the large input. We ignored his submission on the small input, as we do not know whether the participant wrote a separate slower program for the small input or just used his efficient solution. If a contestant only solved the small input, we add this submission to the set of slow programs.

Note that this is a potential source of errors, as the contestant might have in fact written a fast solution, but just did not have the time to run it on both inputs. We explored the possibility of running and timing the submissions ourselves, but that proved to be too challenging, as many contestants expected specific filenames for input and output or made other assumptions of the way they were compiled and run. Some contestants even prepared their code to link it against a custom framework such that they can run each test case in parallel. Also it is not guaranteed, that the participants really upload their proper source code, since it is not checked or in any way enforced by the contest grading system.

We did not consider the easiest task of the qualification round called 'tictactoe', because all but 213 of the 8450 C++-contestants solved the full task, as the large input was not significantly more complicated to solve.

Table 1 list the size of these datasets. We can see that in both rounds the tasks get increasingly harder, such that there are fewer successful submissions and the percentage of slow submissions increases. This way we ended up with a dataset for 6 tasks with a total of 18606 submissions consisting of 1.275 million lines of code.

Feature Extraction

We wrote the python-tool `featureExtraction.py` to run across the dataset and extract 35 integer-valued features for each submission. Inspired by the very few simple features used in the programming style fingerprint in [1], we manipulate the source string in `featureCodeAnalysis.py` to get these features.

Table 1: List of the 6 tasks from Google Code Jam 2013 that we used.

Contest Round	Taskname	# slow samples	# fast samples
Qualification	Lawnmower	305	5738
	Fairsquare	2819	4122
	Treasure	651	118
Round 1A	Bullseye	1127	1391
	Energy	944	520
	Luck	625	246

Table 2: List of the 35 features we extracted

length	lines	includesCount	includesLength	definesCount
definesLength	commentsLength	commentsCount	loops	vars
int	float	double	char	long
depth	deepLength	deepCount	for	while
if	else	switch	maxConstant	functionCount
set	priority_queue	map	multimap	queue
stack	list	vector	unordered_map	unordered_set

We strip defines, includes and comments from the code and count the number of occurrences of several keywords and C++ STL classes in the remaining code. We also extract all integer values in the code and use the maximal value as a feature, which might indicate the biggest bound used in the code. To focus on the efficiency of the implementations we look for the most nested part of the code, so the section of the code where the most curly braces are open, and take some features of this section into account. Table 2 lists all 35 features we used.

We then wrote `featureCollection.py`, which splits our dataset into equally sized training sets and test sets for each task. In order to use our classifier models we need each feature to be binary. All our features are positive integers so far, so we need to convert them by comparing them to some threshold value. We tried out several ways to get such a threshold by taking the following values of the training data:

- median
- mean
- maximum separator (value where the percentage of slow programs versus the percentage of fast programs on one side of the threshold differed the most)
- quartiles (25th, 50th and 75th percentiles)
- 11-quantiles (such that we get all $\frac{100i}{11}$-th percentiles for $i = 1, .., 10$)

This way we converted each integer feature into a single binary feature (for median, mean and maximum separator) or into 3 or 10 correlated binary features.

The median has the advantage that roughly half of the binary features will be zero and half of them will be one. However it is problematic for features that are zero (or any other fixed value) in most of the samples. For instance in the training data of task 'Bullseye' the feature 'set' is non-zero only in 16 out of 1233 samples. So the median is zero and we need to use $>$-comparison (and not $\geq$-comparison) to make sure that only these 16 samples get assigned the binary feature one.

Our script finally outputs MATLAB-codefiles for each dataset. For the multi-task setup we combined the 3 tasks from the qualification round to our multi-task training set and the 3 tasks from the round 1A to our multi-task training set (see script `prepareMultitaskSets.m`). To get v

Single-Task Classifiers

Given these datasets we implemented, trained and applied classifiers for each task separately. We implemented the Naive Bayes classifier (see `nb_learn.m` and `nb_run.m`) and logistic regression classifier (see `logreg_learn.m` and `logreg_run.m`) using what we learned in class and in the first homework.

Multi-Task Classifier

Finally we implemented the multi-task logistic regression model from [3]. We briefly summarize this method here as we did in the literature survey.

Problem setup We are given a set of related binary tasks $T_1, \dots, T_M$ with corresponding training data $D = \{S_1, \dots, S_M\}$. Each dataset consists of some feature vector x of a fixed length d and a binary class y, so $S_i = \{(x_n^i, y_n^i)\}_{n=1,\dots,N(i)}$ with $x_n^i \in \{0, 1\}$ and $y_n^i \in \{0, 1\}$.

If we now train separate logistic regression classifiers f_i for each task T_i we would learn separate weight vectors $\mathbf{w}^{(i)}$ leading to a weight matrix $W = (\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(M)})$. Each classifier f_i uses the same logistic regression formula we saw in class, so

$$f_i(x) = P(y = 1 \mid x, T_i) = \frac{1}{1 + exp(-\mathbf{w}^{(i)} x^T)}.$$

They denote the negated log-likelihood estimator over all tasks as $L(D, W)$ and add a Gaussian prior to regularize the weights. If we just take the norm of the weight matrix we end up with a loss function $H(W)$ that corresponds to what we saw in class

$$H(W) = L(D, W) + \frac{1}{\sigma^2}\|W\|_2, \text{ where } L(D, W) = -\sum_{i=1}^{M}\sum_{n=1}^{N(i)} log(P(y_i^n \mid x_i^n, W)).$$

This would keep the different tasks completely unrelated. To couple the parameters they impose a prior distribution on the rows of W. So instead of the L2-norm of W as the regularization term, they take the norm of the *row-mean vector* $\bar{\mathbf{w}}$ of W into account, where $\bar{\mathbf{w}}_j = \sum_{i=1}^{M} w_j^{(i)}/M$. By adding the deviations of each $\mathbf{w}^{(i)}$ from $\bar{\mathbf{w}}$ to the regularization term, the goal is to get as $\bar{\mathbf{w}}$ the best possible fit across all tasks. They call the model hierarchical as now $\bar{\mathbf{w}}$ is the best fit across all $\mathbf{w}^{(i)}$ and each $\mathbf{w}^{(i)}$ is the best fit for its task. So they propose the loss function

$$G(W) = L(D, W) + R(W), \text{ where } R(W) = \frac{\lambda_1}{2}\|\bar{\mathbf{w}}\|_2 + \frac{\lambda_2}{2}\sum_{i=1}^{M}\|\mathbf{w}^{(i)} - \bar{\mathbf{w}}\|_2.$$

Note that we now have two variances λ_1 and λ_2 that are parameters of the model.

Training Algorithm To train the model, so to minimize the loss function, they applied the BFGS gradient descent method. Compared to what we did in our homework, the derivation of the gradient gets more complicated, as the regularization part of the gradient depends on $\bar{\mathbf{w}}$ now. They derive

$$\frac{\partial R(W)}{\partial w_k^{(s)}} = \lambda_1 \bar{\mathbf{w}}_k \frac{\partial \bar{\mathbf{w}}_k}{\partial w_k^{(s)}} + \lambda_2 \sum_{i=1}^{M}(w_k^{(i)} - \bar{\mathbf{w}}_k)(\delta_{is} - \frac{\partial \bar{\mathbf{w}}_k}{\partial w_k^{(s)}}), \text{ where } \frac{\partial \bar{\mathbf{w}}_j}{\partial w_k^{(s)}} = \begin{cases} \frac{\lambda_2}{\lambda_1 + M\lambda_2} & \text{if } j = k \\ 0 & \text{if } j \neq k \end{cases}.$$

Unfortunately the authors do not mention whether this extended model has an effect on the number of iterations needed until the gradient descent method converges. Also they do not state how to choose the model parameters λ_1 and λ_2. We also tried to calculate the gradient by using the definition of mean weight directly.

$$\frac{\partial R(W)}{\partial w_k^{(s)}} = \frac{\lambda_1}{M}\bar{\mathbf{w}}_k + \lambda_2 \sum_{i=1}^{M}(w_k^{(i)} - \bar{\mathbf{w}}_k)(\delta_{is} - \frac{1}{M})$$

3 Results

3.1 Single-Task Classifiers

We ran Naive Bayes and logistic regression on all 6 tasks and computed accuracy, precision, recall and the F1-measure of each classifier.

3.2 Quality

4 Discussion

Features a static analyzer might bring more and better features

still surprising how good already the basic features work

References

[1] Elenbogen, Bruce S., and Naeem Seliya. ”Detecting outsourced student programming assignments.” Journal of Computing Sciences in Colleges 23.3 (2008): 50-57.

[2] Kremenek, Ted, et al. ”From uncertainty to belief: Inferring the specification within.” Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association, 2006.

[3] Lapedriza, gata, David Masip, and Jordi Vitri. ”A hierarchical approach for multi-task logistic regression.” Pattern Recognition and Image Analysis. Springer Berlin Heidelberg, 2007. 258-265.