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	difficulty
Qualification	Lawnmower	305	5738	easy
	Fairsquare	2819	4122	medium
	Treasure	651	118	hard
Round 1A	Bullseye	1127	1391	easy
	Energy	944	520	medium
	Luck	625	246	hard

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 $T_1, \ldots, T_M$ with corresponding training data $D = \{S_1, \ldots, S_M\}$. Each dataset consists of some feature vector x of a fixed length d and a binary class y, so $S_i = \{(x_n^i, y_n^i)\}_{n=1,\ldots,N(i)}$ 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 $W = (\mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(M)})$. 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

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

Naive Bayes Performances

Naive Bayes Accuracy

	Median	Mean	3 Quantiles	10 Quantiles
Bullseye	70.2229	69.6656	71.7357	71.258
Energy	76.9126	75.9563	79.0984	78.5519
Luck	69.8598	69.1589	67.757	67.2897
Lawnmower	94.2072	95.0348	78.4177	64.5482
Fairsquare	83.6264	69.5013	81.3203	79.8501
Treasure	82.9787	81.9149	83.2447	81.383

Naive Bayes Precision

	Median	Mean	3 Quantiles	10 Quantiles
Bullseye	73.1466	74.1197	75	74.7475
Energy	65.8621	65.343	67.5325	66.9935
Luck	52.6718	51.6667	49.3243	48.7342
Lawnmower	95.0234	95.0348	96.8737	97.4183
Fairsquare	85.6659	76.902	85.0882	85.1309
Treasure	49.3671	46.4789	50	46.3918

Naive Bayes Recall

	Median	Mean	3 Quantiles	10 Quantiles
Bullseye	67.7863	64.2748	68.7023	67.7863
Energy	73.1801	69.3487	79.6935	78.5441
Luck	50.7353	45.5882	53.6765	56.6176
Lawnmower	99.0944	100	79.8676	64.4026
Fairsquare	86.6732	68.8388	82.7536	79.6668
Treasure	61.9048	52.381	73.0159	71.4286

Naive Bayes F1 Measure

	Median	Mean	3 Quantiles	10 Quantiles
Bullseye	70.3645	68.8471	71.7131	71.0969
Energy	69.3285	67.2862	73.1107	72.3104
Luck	51.6854	48.4375	51.4085	52.381
Lawnmower	97.0162	97.4542	87.5525	77.5425
Fairsquare	86.1666	72.6474	83.9046	82.3083
Treasure	54.9296	49.2537	59.3548	56.25

Figure 1: Performance of the Naive Bayes classifier.

We first trained and run Naive Bayes using the different thresholds. Figure 1 gives all the results for the Naive Bayes classifier. As expected, we found that the Naive Bayes classifier can not profit from multi-bit features, as they clearly violate the independence assumption. In the task lawnmower, where 95% of all submissions solve the large testcase, the multi-bit versions perform significantly worse than the median- and mean-versions. In the other tasks the differences were smaller, so we used the median threshold for further experiments, as it performed best on average.

Logistic Regression Figre 2 summarizes the result on logistic regression for different regularization parameters $\lambda \in [10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}]$. We used the quartile-thresholds for this experiment, where each integer feature is represented using 3 bits. For the four easier tasks the results were almost identical. Only in the two hard tasks *Luck* and *Treasure* the choice of $\lambda = 10^{-3}$ works best. So we used this value in the later comparison.

Single-Task Comparison Next, we compared the following four classifiers:

- Naive Bayes using the median as threshold and only the first 5 features (code length in bytes and lines, number of includes, number of loops and number of variables)
- Naive Bayes using the median as threshold and all 35 features

Logistic Regression Performances

Logistic Regression Accuracy

	λ = 10^-2	λ = 10^-3	λ = 10^-4	λ = 10^-5
Bullseye	78.2643	78.5032	77.8662	77.8662
Energy	82.2404	82.5137	82.1038	82.2404
Luck	74.5327	73.8318	73.1308	73.1308
Lawnmower	95.0348	95.0679	94.9355	94.8361
Fairsquare	86.019	85.9902	85.8749	85.8461
Treasure	85.6383	84.5745	81.9149	80.0532

Logistic Regression Precision

	λ = 10^-2	λ = 10^-3	λ = 10^-4	λ = 10^-5
Bullseye	76.9774	77.5393	77.0445	77.0445
Energy	75.6863	75.8755	75.5906	75.6863
Luck	68.4932	63.0435	60.8247	60.8247
Lawnmower	95.0348	95.0961	95.1795	95.1747
Fairsquare	86.9421	87.506	87.4457	87.4396
Treasure	66.6667	55.5556	45.4545	40.3226

Logistic Regression Recall

	λ = 10^-2	λ = 10^-3	λ = 10^-4	λ = 10^-5
Bullseye	83.2061	82.7481	81.9847	81.9847
Energy	73.9464	74.7126	73.5632	73.9464
Luck	36.7647	42.6471	43.3824	43.3824
Lawnmower	100	99.9652	99.7214	99.6169
Fairsquare	89.7109	88.878	88.731	88.682
Treasure	28.5714	39.6825	39.6825	39.6825

Logistic Regression F1 Measure

	λ = 10^-2	λ = 10^-3	λ = 10^-4	λ = 10^-5
Bullseye	79.9707	80.0591	79.4379	79.4379
Energy	74.8062	75.2896	74.5631	74.8062
Luck	47.8469	50.8772	50.6438	50.6438
Lawnmower	97.4542	97.4699	97.3975	97.3451
Fairsquare	88.3048	88.1867	88.0837	88.0564
Treasure	40	46.2963	42.3729	40

Figure 2: Performance of the logistic regression classifier.

Single-Task Performances

Single Task Accuracy

	NB Median 5 Features	NB Median 35 Features	Log Reg Median	Log Reg 3 Quantiles
Bullseye	59.793	70.2229	75.7962	78.5032
Energy	69.5355	76.9126	76.3661	82.5137
Luck	71.2617	69.8598	69.6262	73.8318
Lawnmower	95.0348	94.2072	94.5382	95.0679
Fairsquare	61.6604	83.6264	84.5489	85.9902
Treasure	80.3191	82.9787	83.5106	84.5745

Single Task Precision

	NB Median 5 Features	NB Median 35 Features	Log Reg Median	Log Reg 3 Quantiles
Bullseye	64.9402	73.1466	74.0082	77.5393
Energy	57.197	65.8621	65.942	75.8755
Luck	56.701	52.6718	53.9474	63.0435
Lawnmower	95.0348	95.0234	95.0399	95.0961
Fairsquare	67.4693	85.6659	86.1953	87.506
Treasure	41.791	49.3671	51.8519	55.5556

Single Task Recall

	NB Median 5 Features	NB Median 35 Features	Log Reg Median	Log Reg 3 Quantiles
Bullseye	49.771	67.7863	82.5954	82.7481
Energy	57.8544	73.1801	69.7318	74.7126
Luck	40.4412	50.7353	30.1471	42.6471
Lawnmower	100	99.0944	99.4427	99.9652
Fairsquare	67.2709	86.6732	87.8001	88.878
Treasure	44.4444	61.9048	22.2222	39.6825

Single Task F1 Measure

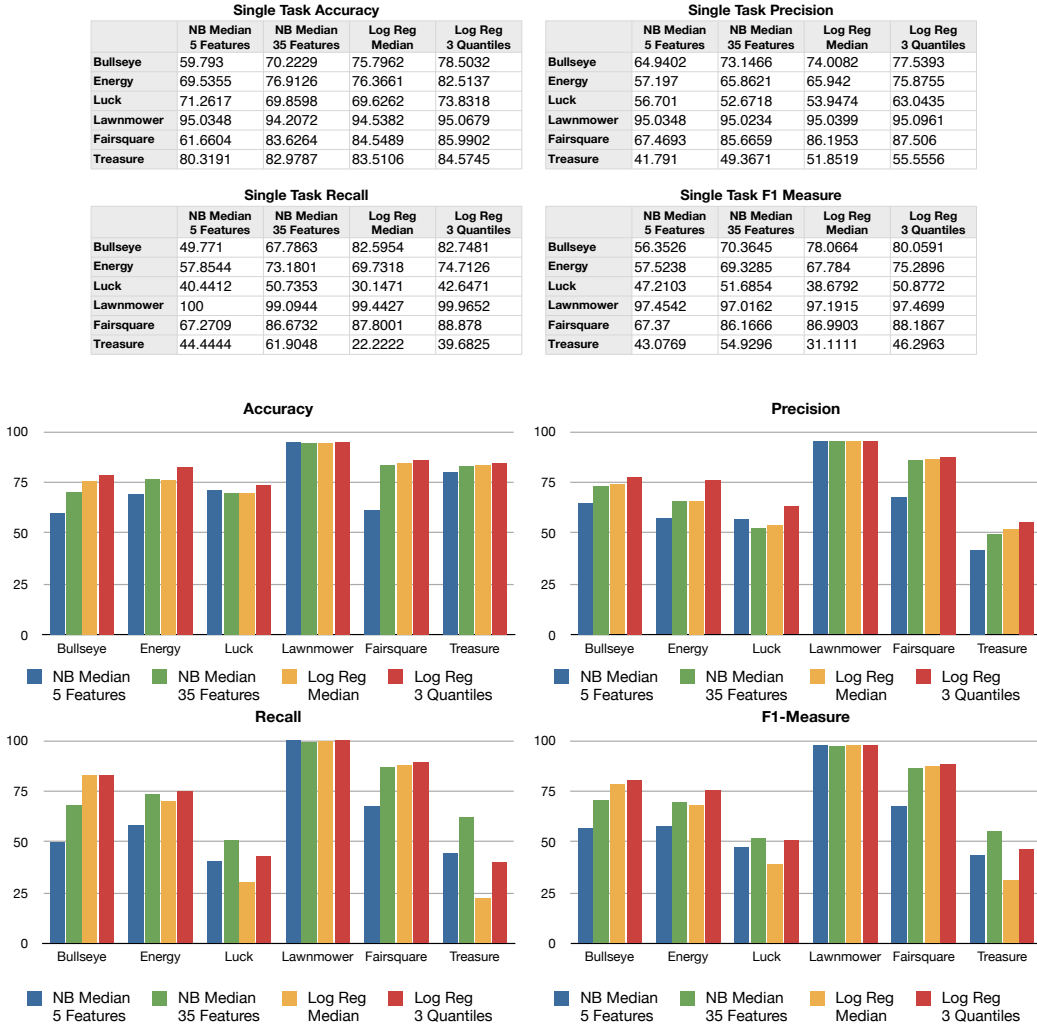|	NB Median 5 Features	NB Median 35 Features	Log Reg Median	Log Reg 3 Quantiles
Bullseye	56.3526	70.3645	78.0664	80.0591
Energy	57.5238	69.3285	67.784	75.2896
Luck	47.2103	51.6854	38.6792	50.8772
Lawnmower	97.4542	97.0162	97.1915	97.4699
Fairsquare	67.37	86.1666	86.9903	88.1867
Treasure	43.0769	54.9296	31.1111	46.2963

Figure 3: Comparison of the various single-task classifiers.

- Logistic regression using a single bit per feature from the median threshold
- Logistic regression using three bits per feature from the quartile thresholds

The results are listed in Figure 3. We can see that the three-bit logistic regression outperforms the Naive Bayes in the four easier tasks, but falls behind it in the four harder tasks.

3.2 Multi-Task Classifiers

All the data, code and results of our experiments can be found on our GitHub-repository (link).

4 Discussion

Features a static analyzer might bring more and better features

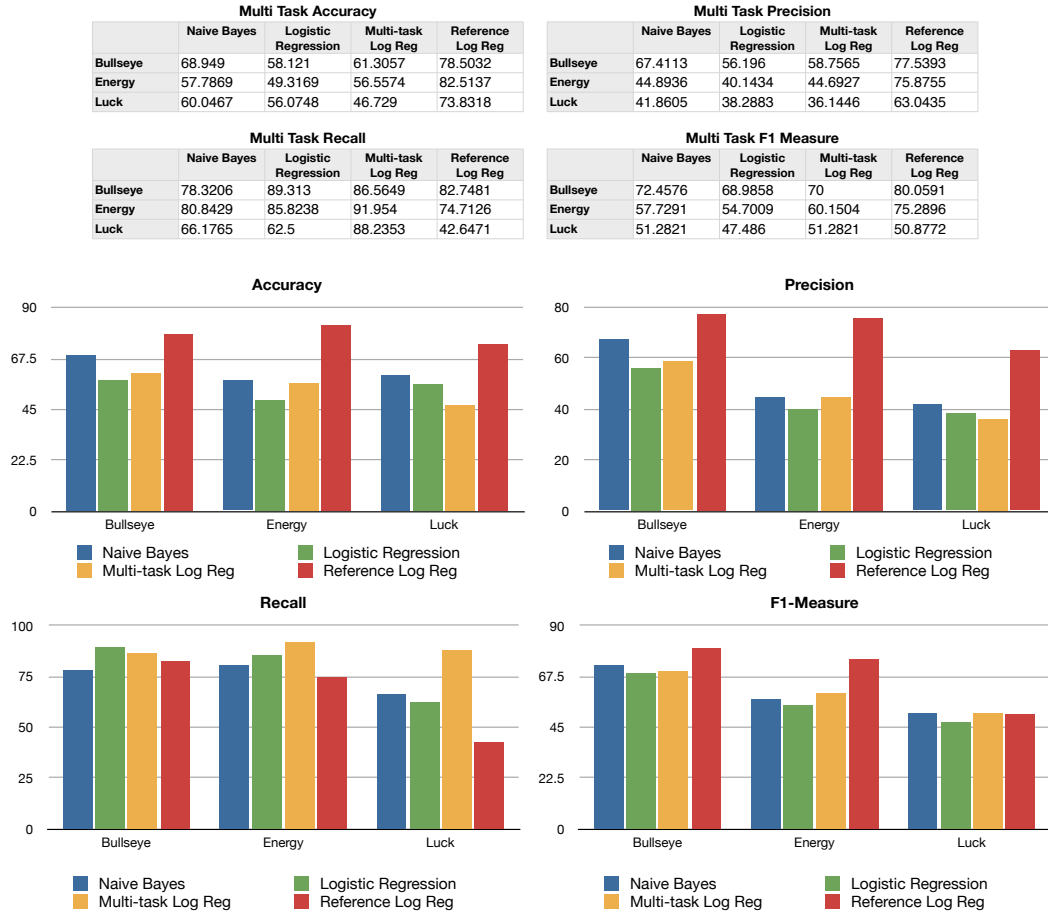still surprising how good already the basic features work

Multi-Task Performances

Multi Task Accuracy

	Naive Bayes	Logistic Regression	Multi-task Log Reg	Reference Log Reg
Bullseye	68.949	58.121	61.3057	78.5032
Energy	57.7869	49.3169	56.5574	82.5137
Luck	60.0467	56.0748	46.729	73.8318

Multi Task Precision

	Naive Bayes	Logistic Regression	Multi-task Log Reg	Reference Log Reg
Bullseye	67.4113	56.196	58.7565	77.5393
Energy	44.8936	40.1434	44.6927	75.8755
Luck	41.8605	38.2883	36.1446	63.0435

Multi Task Recall

	Naive Bayes	Logistic Regression	Multi-task Log Reg	Reference Log Reg
Bullseye	78.3206	89.313	86.5649	82.7481
Energy	80.8429	85.8238	91.954	74.7126
Luck	66.1765	62.5	88.2353	42.6471

Multi Task F1 Measure

	Naive Bayes	Logistic Regression	Multi-task Log Reg	Reference Log Reg
Bullseye	72.4576	68.9858	70	80.0591
Energy	57.7291	54.7009	60.1504	75.2896
Luck	51.2821	47.486	51.2821	50.8772

Figure 4: Comparison of the various multi-task classifiers.

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.