Structured Gotos are (Slightly) Harmful

Eli Sennesh
Technion
Technion City
Haifa, Israel
esennesh@cs.technion.ac.il

Yossi Gil
Technion
Technion City
Haifa, Israel
yogi@cs.technion.ac.il

ABSTRACT

We take up the questions of if and how "structured **goto**" statements impact defect proneness, and of which what concept of size yields a superior metric for defect prediction.

We count **goto**-like unstructured jumps, alongside method size and compressed method size, as software engineering metrics, and examine the evolution of 26 open-source code corpora in relation to those metrics. We employ five different measures of defectiveness and development effort. We measure the statistical quality of our metrics as predictors of our defect measurements.

We show that the number of unstructured jumps is a predictor of defects, routine maintenance and two other metrics of software development effort. The correlation between unstructured jumps and development effort is positive, and it remains so even after accounting for the effect of code size. We also show that the number of unstructured jumps is superior to code size, both compressed and uncompressed, in its predictive power of accumulated defects.

CCS Concepts

•**Software and its engineering** → **Control structures; Software evolution;** *Software defect analysis;*

Keywords

software defect prediction, static code metrics, control-flow constructs

1. INTRODUCTION

Dijkstra advocated eliminating **goto** statements from code as early as 1968 [4], holding that its usage

> "*has an immediate consequence that it becomes terribly hard to find any meaningful set of coordinates in which to describe the process progress*."

The case against **goto** usage is not limited to mere argumentation. **goto** increases a program's cyclomatic complexity [9]. It makes control-flow graphs irreducible [1], thereby complicating static analysis and compiler-level optimization of the code. **goto** statements are also formally unnecessary, as demonstrated by Böhm [3], whose *structured programming theorem* shows how to take any program using **goto** and construct an equivalent but structured program without **goto**.

The primary advantage of **goto** over higher-level control-flow constructs is its simpler translation into single, unconditional branch instructions, and thus its greater efficiency. Knuth [7] proposed using **goto** for exactly this purpose, arguing that certain uses of **goto** are in harmony with structured programming.

This argument by Knuth to allow **goto** in disguise seems to have won out. While **goto** itself rarely appears in most modern code, the constructs **break**, **continue**, **return** and **throw** routinely do. These violate the fundamental principle of structured programming: that every command, atomic or compound, has precisely one entry point and (no more than) one exit point.

Consider the block of code in Listing 1, paraphrased from one of these authors' own codebases (but with trademarked names removed and variable names changed). The seemingly-infinite loop structure might make sense in, for instance, an autonomous worker-thread that moves bytes in parallel to other processing, but in fact this code forms part of a procedure for synchronously replaying old network traffic. The pattern of setting `retval` and then issuing a **break** command appears twice, but *could* have appeared in the first **if** body as well.

Is this code correct? To the authors' knowledge, it doesn't have an outstanding bug report at this time, but that's no proof (we've found bugs in code of similar age and frequency of usage). A quick `grep` search shows that the **while** `(1)` control-flow trick appears at least twelve times in just a single revision of a single `git` repository.

Following the drive ignited by [8] towards evidence-based language design and software engineering, we take up whether and how "structured **goto**" statements impact defect proneness, and which concept of size yields a superior metric for defect prediction. Rather than harangue, our contribution is to treat these "structured **goto**" statements as a metric and analyze empirically their correlation with other empirical measures of defects.

SAC 2016,April 04 - 08, 2016, Pisa, Italy

ACM 978-1-4503-3739-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2851613.2851730

```
while (1) {
  nread = net_tape->read(fd, &hdr, sizeof(
    hdr));
  if (!nread)
    break;

  if (nread != sizeof(hdr)) {
    net_error("net-tape: invalid byte count
    read, aborting...\n");
    retval = -EIO;
    break;
  }

  nread = net_tape->read(fd, buffer, hdr.
    size);
  if (hdr.size != nread) {
    net_error("net-tape: invalid byte count
    read, aborting...\n");
    retval = -EIO;
    break;
  }

  net_rx_handler(hdr.cport, buffer, nread);
}
```

Listing 1: Unstructured jumps in industrial code

The advent of easier to parse languages such as Java, and the availability on the Internet of open-source repositories, along with their history, made our study not only possible, but also feasible.

Contributions.

We here use the term *unstructured jumps* to denote jump instructions traversing more than one edge of the abstract-syntax tree, or that leave blocks of code with more than one entry or exit point. An intuitive way to think of these is as "hidden **goto**'s". We investigate their use in a dataset comprised of professionally-developed software projects.

Our specific contributions are as follows:

1. We show that the number of unstructured jumps is a predictor of defects, routine maintenance and two other metrics of software development effort.
2. We show that the correlation between unstructured jumps and development effort is positive.
3. We show that this correlation *remains* positive even after the confounding variable of code size is factored out.
4. We find that the number of unstructured jumps is (minutely) superior to code size in its correlation with code defects.

2. METRICS AND STATISTICS

2.1 Independent Variables: Code Metrics

We employed an unstructured-jump metric (**USJ**), a program-size metric (**NOT**), and a compressed size metric (**GZP**). We also included a control metric containing randomly-generated numbers (**MNK**), which was expected to correlate significantly with defects only at the alpha level (for example, 5% of the time when $\alpha = 0.05$).

1. *Unstructured Jumps* (**USJ**): a count of all **return** statements outside tail-position, all **break** and **continue** statements within loops, all infix Boolean operators with short-circuit evaluation, and all **throw** statements found within each method.
2. *Number of Tokens* (**NOT**): The number of tokens in each method, representing method size. This metric was preferred over the more traditional lines of code (LOC) for being robust to formatting conventions and the presence of comments.
3. *GZIP* (**GZP**): The compressed size of each method, measured in bytes of gzipped source code.
4. *Monkey Metric* (**MNK**): A randomly generated real number, used for control and sanity check.

We employed the number of tokens (**NOT**) in a method as our length metric rather than the number of lines of code it contains due to the greater robustness of tokens over lines against different coding and formatting styles.

It is a commonly held view, though mostly falsified by [5], that the single strongest predictor of defect proneness is a function's length. On this basis, we also normalized our metric values in three different ways to remove the effect of method length upon them:

1. *Size Normalization*: metric values for a method are divided by the method-length metric (**NOT**) value at that method and revision
2. *Rank Normalization*: metric values for each method at each revision are transformed into ranks, and each metric-value rank is divided by the corresponding method-length metric (**NOT**) rank for the same method and revision
3. *Compressed Size Normalization*: metric values for a method are divided by the compress-size metric (**GZP**) value at that method and revision

2.2 Dependent Variables: Metrics for Development Effort

Unfortunately, the overwhelming majority of available software corpora do not include bug-tracking data, and actually existing bug reporting is not always accurate. We therefore employed five different measurements for development effort, relying on their consensus to satisfactorily approximate real defect rates.

The analysis was method-based (rather than file-, or class-based).

1. Defect *Proneness*: whether or not a revision under examination had a commit message matching a regular expression which searches for words such as "fix" and "bug" case-insensitively, as well as numbers preceded by #-signs (to denote bug-report numbers). *Proneness* provides a direct way of measuring the presence of defects, even if it always undercounts relative to human assessments of defect presence [6].

2. *Defects*: the accumulated number of times a revision under examination had a commit message matching a regular expression which searches for words such as "fix" and "bug" case-insensitively, as well as numbers preceded by #-signs (to denote bug-report numbers). *Defects* provides a direct way of measuring the quantity of defects, even if it must necessarily undercount.

3. *Versions*: the accumulated number of times a method's source code was changed. *Versions* provides a measurement of how often development effort had to be expended on a method.

4. *Churn*: the accumulated lines of code changed in a method's source code, inspired directly by the work of [10], in which relative churn was found to be a good predictor of defects. *Churn* also measures how much development effort had to be expended on a method.

5. *Maintenance*: the accumulated lines of code changed in defective revisions, effectively a relative-churn metric for only those methods with boolean-true defect *Proneness*. Similarly to *Proneness* and *Defects*, *Maintenance* necessarily undercounts defect presence.

These measures of software evolution are computable directly from `git` logs, and therefore represent phenomena which were visible to the programmers who ordered the commits in the first place. By comparing defect measures in relation to metrics, we approximate the relationship between those metrics and the true defect rate, despite the lack of reliable direct defect reports. The *Churn* measurement in particular was inspired directly by the work of Nagappan [10], in which relative churn was found to be a good predictor of defects.

Take note that as usual in statistical studies of an existing population, these variables are not strictly independent, e.g., code size is obviously driven by factors such as development culture, individual style, etc. In addition, defect rates detected or predicted statically, using metrics, have been found to underestimate real defect rates [6].

2.3 Distribution of Code Metrics

Table 1: p-values from the Kolmogorov-Smirnov test of uniformity for metric values

	USJ	NOT	GZP	MNK
$p \geq 0.05$	0	0	0	26
$p < 0.05$	0	0	0	0
$p < 0.01$	0	0	0	0
$p < 0.001$	26	26	26	0

We applied our code metrics to individual methods rather than to whole JAVA [**?**] source files. [2] previously observed that metric values tend to be distributed neither normally nor uniformly. We confirmed this by performing the Kolmogorov-Smirnov test of uniformity on all metric values, of which the results are displayed in Table 1. **`USJ`**, **`NOT`**, and **`GZP`** all reject the null hypothesis of uniformity in all corpora, with $p < 0.001$. The only metric which does not reject the null hypothesis of uniformity is **`MNK`**, defined as a real number uniformly sampled from the interval $[0, 1]$.

3. EMPIRICAL FINDINGS

3.1 Preliminary χ^2 Tests

We performed a χ^2 test of independence to see the likelihood that defect *Proneness* is conditionally independent from all our code metrics, displaying the results in Table 2.

Table 2: p-values for the χ^2 test of independence between methods with defect *Proneness* of 1 and methods with defect *Proneness* of 0

	USJ	NOT	GZP	MNK
$p \geq 0.05$	0	3	1	23
$p < 0.05$	0	1	1	2
$p < 0.01$	0	0	0	1
$p < 0.001$	26	22	24	0

Under the χ^2 test's null hypothesis, defect *Proneness* and code metrics have no relation, and defect-prone methods should thus exhibit the same distributions of metric values as non-defective methods. If we reject the null hypothesis, the alternative is that defective and nondefective methods have significantly different distributions of metric values.

The table shows the following. Defect *Proneness* in all corpora presents a very strong significant relationship with **`USJ`**. 22 showed very strong significance (and 1 regular significance) against **`NOT`**, and 24 showed very strong significance (and 1 regular significance) against **`GZP`**. As expected for $p < 0.05$, two out of the 26 corpora (7.7%) showed a significant relationship with the random **`MNK`** metric. Most of the relationships we found were extremely strong, with $p < 0.001$ being the mode likelihood of the null hypothesis.

3.2 Predictive power of code metrics

Kendall's τ_b is a rank-correlation coefficient that measures the similarity of ordering between two random variables. In paired samples of the form (x_i, y_i) from two random variables, samples are concordant when $x_i \leq x_j$ and $y_i \leq y_j$, discordant when $x_i \leq x_j$ but $y_j \leq y_i$, and neither otherwise. The τ_b coefficient is then defined by subtracting the number of discordant pairs from the number of concordant pairs and dividing by a normalization constant to bring the result between -1 and +1. The τ_b coefficient's distribution has an expected-value of 0, and becomes approximately normal (with mean of 0, again) with large sample sizes. Since our sample size is in the thousands, we employed the normal approximation to perform a hypothesis test for significant deviation from the null hypothesis of no rank-correlation.

Each of our τ_b tables lists corpora as its rows and metrics as its columns, giving per-metric mean τ_b values at the bottom to tell us how well the metric predicted the matching measurement (of code defects or development effort) on average. The values range from -1.0 for deterministic anticorrelation to 1.0 for deterministic correlation.

3.2.1 *Metrics predicting Defects*

Table 3 and Table 4 show the results of measuring Kendall's τ_b between metrics and *Defects* under no normalization and size-normalization.

`USJ` best predicted *Defects*, but only very slightly compared to **`NOT`**. The vast majority of the correlations were statistically signif-

Table 3: Predictability of *Defects* from unnormalized metric values, measured by Kendall's τ_b. Values range between -1.0 and 1.0.

ID	USJ	NOT	GZP	MNK
A	0.2279^{**}	0.2520^{**}	0.2617^{**}	0.0001
B	0.2041^{**}	0.1888^{**}	0.1700^{**}	0.0023
C	0.0681^{**}	0.1022^{**}	0.1202^{**}	−0.0073
D	0.0991^{**}	0.0626^{**}	0.0744^{**}	−0.0004
E	0.1799^{**}	0.1998^{**}	0.1949^{**}	−0.0006
F	0.3270^{**}	0.3999^{**}	0.4065^{**}	0.0025
G	0.0487^{**}	0.0183^{**}	0.0134^{**}	−0.0018
H	0.1124^{**}	0.0768^{**}	0.0633^{**}	−0.0017
I	0.1044^{**}	0.1047^{**}	0.0997^{**}	0.0006
J	0.0660^{**}	0.0525^{**}	0.0438^{**}	0.0024
K	0.0928^{**}	0.1128^{**}	0.1128^{**}	0.0067^{*}
L	0.1859^{**}	0.1590^{**}	0.1436^{**}	−0.0001
M	0.0302^{**}	0.0042	-0.0566^{**}	0.0031
N	0.1768^{**}	0.0698^{**}	0.0059	0.0055
O	0.1416^{**}	0.0919^{**}	0.1248^{**}	0.0077
P	0.0531^{**}	0.0278^{**}	0.0419^{**}	−0.0051
Q	0.1706^{**}	0.2774^{**}	0.2556^{**}	0.0002
R	0.1421^{**}	0.1800^{**}	0.1583^{**}	−0.0099
S	0.1354^{**}	0.0889^{**}	0.1008^{**}	−0.0041
T	0.0826^{**}	0.1781^{**}	0.1793^{**}	0.0001
U	0.1728^{**}	0.1841^{**}	0.1909^{**}	0.0007
V	0.0521^{**}	-0.0913^{**}	-0.0168^{*}	0.0041
W	0.0923^{**}	0.0373^{**}	0.0260^{**}	0.0000
X	0.1143^{**}	0.1742^{**}	0.1690^{**}	−0.0070
Y	0.1997^{**}	0.2468^{**}	0.2438^{**}	−0.0043
Z	0.1140^{**}	0.1899^{**}	0.1912^{**}	0.0026
Metric mean	0.1305	0.1303	0.1276	−0.0001

$^{*}\ p < 0.05$
$^{**}\ p < 0.01$

Table 4: Predictability of *Defects* from size-normalized metric values, measured by Kendall's τ_b. Values range between -1.0 and 1.0.

Size	USJ	GZP	MNK
A	0.1588^{**}	-0.1937^{**}	-0.2009^{**}
B	0.1741^{**}	-0.1991^{**}	-0.1557^{**}
C	0.0539^{**}	-0.0590^{**}	-0.0793^{**}
D	0.0835^{**}	-0.0359^{**}	-0.0551^{**}
E	0.1563^{**}	-0.1761^{**}	-0.1645^{**}
F	0.1645^{**}	-0.3522^{**}	-0.3209^{**}
G	0.0485^{**}	-0.0187^{**}	-0.0181^{**}
H	0.1019^{**}	-0.0642^{**}	-0.0598^{**}
I	0.0856^{**}	-0.0881^{**}	-0.0839^{**}
J	0.0547^{**}	-0.0595^{**}	-0.0406^{**}
K	0.0686^{**}	-0.0829^{**}	-0.0861^{**}
L	0.1755^{**}	-0.1407^{**}	-0.1264^{**}
M	0.0290^{**}	-0.1128^{**}	−0.0013
N	0.1740^{**}	-0.1684^{**}	-0.0518^{**}
O	0.1254^{**}	-0.0200^{*}	-0.0701^{**}
P	0.0484^{**}	0.0047	-0.0247^{*}
Q	0.1070^{**}	-0.2720^{**}	-0.2334^{**}
R	0.0832^{**}	-0.1784^{**}	-0.1458^{**}
S	0.1074^{**}	-0.0489^{**}	-0.0673^{**}
T	0.0409^{**}	-0.1306^{**}	-0.1411^{**}
U	0.1501^{**}	-0.1062^{**}	-0.1449^{**}
V	0.0520^{**}	0.0960^{**}	0.0709^{**}
W	0.0880^{**}	-0.0379^{**}	-0.0285^{**}
X	0.0903^{**}	-0.1389^{**}	-0.1448^{**}
Y	0.1608^{**}	-0.2054^{**}	-0.2012^{**}
Z	0.0856^{**}	-0.1455^{**}	-0.1515^{**}
Metric mean	0.1026	−0.1129	−0.1049

$^{*}\ p < 0.05$
$^{**}\ p < 0.01$

icant, with **MNK** showing a significant ($p < 0.05$) correlation only once among all 26 corpora.

Size-normalizing the metrics added information to their values from **NOT**, which explains their all maintaining or even gaining statistical significance, even **MNK**. **USJ** maintains a noticeable mean correlation with *Defects*, while **GZP** and **MNK** had their information content dominated by that of **NOT** and became anticorrelated with *Defects*.

3.2.2 *Metrics predicting* Churn

Table 5 and Table 6 show the results of measuring Kendall's τ_b between metrics and *Churn* under no normalization and size-normalization.

GZP most strongly predicted *Churn*, followed closely by **NOT** and then, with a lower mean τ_b by nearly 0.10, **USJ**. All correlations were significant in all corpora, except for those with **MNK**, for which $p < 0.05$ was obtained only twice in 26 corpora.

Size-normalization once again resulted in **USJ** being the only metric to hold on to positive correlation rather than becoming dominated by **NOT**'s information content: **USJ** had a positive and substantial mean τ_b after size normalization while all other metrics anticorrelated. Statistical significances were again maintained, and added to **MNK** by the information content of the normalization.

3.2.3 *Metrics predicting* Maintenance

Table 7 and Table 8 show the results of measuring Kendall's τ_b between metrics and *Maintenance* under no normalization and size-normalization.

GZP showed the highest average correlation with *Maintenance*, followed by **NOT** and **USJ**. **USJ**'s correlations were statistically significant less often than those of **NOT** and **GZP**. None of the metrics had a larger τ_b value with *Maintenance* than 0.10.

Size-normalization again found **GZP** and **MNK** to anticorrelate with *Maintenance*, although **USJ** merely lost some statistical significances while maintaining a low but positive correlation.

4. CONCLUSIONS

We observed strongly significant evidence in the χ^2 test for a relationship between unstructured jumps and the presence or absence (but not quantity) of *Defects*, but also for a relationship between program size and compressed size and defects (subsection 3.1). However, very large sample sizes yield very high power in statistical hypothesis tests; this can lead to very small effects becoming significant. The randomized **MNK** metric having achieved significance twice in the χ^2 test shows that this may have occurred in our experiment.

In subsubsection 3.2.1 we measured the ability of metrics to predict *Defects*. Our correlation measurements found **USJ** to be, slightly but significantly, the strongest predictor of *Defects*, and to lose only 0.0279 points of correlation under size-normalization while

Table 5: Predictability of *Churn* from unnormalized metric values, measured by Kendall's τ_b. Values range between -1.0 and 1.0.

ID	USJ	NOT	GZP	MNK
A	0.3299^{**}	0.4073^{**}	0.4223^{**}	-0.0017
B	0.2398^{**}	0.2266^{**}	0.1937^{**}	-0.0032
C	0.1801^{**}	0.3026^{**}	0.3033^{**}	-0.0160
D	0.1287^{**}	0.1230^{**}	0.1300^{**}	-0.0069
E	0.2111^{**}	0.3241^{**}	0.3124^{**}	0.0012
F	0.4153^{**}	0.5341^{**}	0.5350^{**}	-0.0035
G	0.1734^{**}	0.1377^{**}	0.1452^{**}	-0.0062
H	0.2184^{**}	0.3960^{**}	0.3966^{**}	-0.0033
I	0.2160^{**}	0.3290^{**}	0.3281^{**}	-0.0007
J	0.1244^{**}	0.1986^{**}	0.1925^{**}	0.0033
K	0.2100^{**}	0.3588^{**}	0.3557^{**}	-0.0027
L	0.3052^{**}	0.4437^{**}	0.4417^{**}	0.0065
M	0.1451^{**}	0.2038^{**}	0.1452^{**}	0.0001
N	0.1786^{**}	0.1941^{**}	0.1439^{**}	0.0035
O	0.2215^{**}	0.3948^{**}	0.4396^{**}	-0.0044
P	0.1996^{**}	0.3031^{**}	0.3239^{**}	-0.0175^{*}
Q	0.3433^{**}	0.5422^{**}	0.5234^{**}	0.0054
R	0.3936^{**}	0.5559^{**}	0.5453^{**}	-0.0084
S	0.2614^{**}	0.3632^{**}	0.3723^{**}	-0.0139^{*}
T	0.2002^{**}	0.2710^{**}	0.2711^{**}	-0.0008
U	0.3282^{**}	0.3947^{**}	0.4049^{**}	0.0022
V	0.0738^{**}	0.1583^{**}	0.3188^{**}	-0.0012
W	0.2402^{**}	0.2959^{**}	0.3260^{**}	-0.0039
X	0.2196^{**}	0.3165^{**}	0.3287^{**}	0.0021
Y	0.3039^{**}	0.5038^{**}	0.5002^{**}	-0.0050
Z	0.2446^{**}	0.3741^{**}	0.3656^{**}	0.0025
Metric mean	0.2348	0.3328	0.3371	−0.0028

$^{*}\ p < 0.05$
$^{**}\ p < 0.01$

Table 6: Predictability of *Churn* from size-normalized metric values, measured by Kendall's τ_b. Values range between -1.0 and 1.0.

Size	USJ	GZP	MNK
A	0.2442^{**}	-0.3069^{**}	-0.3182^{**}
B	0.1969^{**}	-0.2713^{**}	-0.1922^{**}
C	0.1543^{**}	-0.2122^{**}	-0.2177^{**}
D	0.1120^{**}	-0.0324^{**}	-0.0785^{**}
E	0.1792^{**}	-0.2846^{**}	-0.2632^{**}
F	0.2116^{**}	-0.4635^{**}	-0.4211^{**}
G	0.1687^{**}	-0.0467^{**}	-0.0972^{**}
H	0.1913^{**}	-0.2488^{**}	-0.2757^{**}
I	0.1813^{**}	-0.2375^{**}	-0.2471^{**}
J	0.1006^{**}	-0.1731^{**}	-0.1475^{**}
K	0.1584^{**}	-0.2619^{**}	-0.2727^{**}
L	0.2787^{**}	-0.3231^{**}	-0.3175^{**}
M	0.1286^{**}	-0.2456^{**}	-0.1571^{**}
N	0.1749^{**}	-0.2192^{**}	-0.1386^{**}
O	0.1929^{**}	-0.1487^{**}	-0.2845^{**}
P	0.1791^{**}	-0.1124^{**}	-0.2082^{**}
Q	0.2394^{**}	-0.4500^{**}	-0.4267^{**}
R	0.2820^{**}	-0.3966^{**}	-0.4278^{**}
S	0.2016^{**}	-0.2401^{**}	-0.2692^{**}
T	0.1374^{**}	-0.1963^{**}	-0.2104^{**}
U	0.2910^{**}	-0.2124^{**}	-0.2890^{**}
V	0.0670^{**}	0.0338^{**}	-0.1063^{**}
W	0.2197^{**}	-0.1529^{**}	-0.2131^{**}
X	0.1808^{**}	-0.2130^{**}	-0.2393^{**}
Y	0.2371^{**}	-0.3945^{**}	-0.3897^{**}
Z	0.2010^{**}	-0.2971^{**}	-0.2866^{**}
Metric mean	0.1888	−0.2349	−0.2498

$^{*}\ p < 0.05$
$^{**}\ p < 0.01$

all other metrics gain anticorrelation.

Although corpora G, M, P, V, and W showed outlying ($\tau_b < 0.0500$) correlations with **NOT** and **GZP**, these corpora still showed their strongest *Defects*-correlation with **USJ**; P, V, and W still showed $\tau_b \geq 0.0500$ with **USJ**. M, the only corpus to fail a significance test for correlation between **NOT** and *Defects*, still rejected the null hypothesis with $p < 0.01$ when testing the link between **USJ** and *Defects*. Likewise, M and V showed significant anticorrelation between compressed program size and *Defects*, but still both showed significance between *Defects* and **USJ**. As in all other corpora, the links between **USJ** and *Defects* in these corpora are almost entirely maintained under size-normalization.

Overall, it appears that **USJ** provides slight but significant power to predict *Defects*, not only independently from **NOT** but even when **NOT** cannot predict very well itself.

When we measured in terms of *Churn* instead of *Defects* in subsubsection 3.2.2, we find that **GZP** becomes the best predictor, while under size-normalization **USJ** loses only 0.0460 points of its correlation. *A priori*, since *Churn* measures the cumulative number of lines of code that were changed in a method across its lifetime, we expect it to correlate more strongly with size metrics such as **NOT** and **GZP** rather than specific programming constructs like **USJ**. The performance of **USJ** under size-normalization does provide weak evidence in its favor as a predictor, however.

Measuring in terms of *Maintenance* (changed lines in code with nonzero *Defects*) would be expected to again correlate closely with program size or compressed size, and so it did in subsubsection 3.2.3. **GZP** showed the most predictive power against *Maintenance* prior to size-normalization.

P and S were outlier corpora; in the former there were no significant correlations, and in the latter **USJ** showed the largest correlation with *Maintenance* and the only statistical significance. P showed a significant positive correlation between **GZP** and *Maintenance* after the size and rank normalizations, and S showed a link between **USJ** and *Maintenance* after size-normalization.

In contrast to **GZP**, **USJ** kept its positive correlations with *Maintenance* under size-normalization, losing only 0.0092 points of correlation.

Overall, it appears that **goto** may deserve to be "considered harmful". If this conclusion appears trivially intuitive, we still benefit from having empirical evidence in its favor. However, our results are not entirely trivial: instead of finding that **goto** is very strongly harmful (as Dijkstra held) or not harmful at all (as Knuth and others held), we find that it is weakly harmful, but with great statistical significance. We also found, more often than not, that rather than **NOT** having the greatest predictive power, either **USJ** (unstructured jumps) or **GZP** (compressed size) did.

5. ACKNOWLEDGMENTS

Table 7: Predictability of *Maintenance* from unnormalized metric values, measured by Kendall's τ_b. Values range between -1.0 and 1.0.

ID	USJ	NOT	GZP	MNK
A	0.0750^{**}	0.1048^{**}	0.1125^{**}	0.0002
B	0.0780^{**}	0.0886^{**}	0.1050^{**}	0.0007
C	0.0081	0.0335^{**}	0.0399^{**}	0.0110
D	0.0774^{**}	0.0619^{**}	0.0571^{**}	-0.0009
E	0.0502^{**}	0.0639^{**}	0.0653^{**}	0.0071^{*}
F	0.0843^{**}	0.1036^{**}	0.1027^{**}	0.0049
G	0.0071	0.0802^{**}	0.0800^{**}	0.0065
H	0.0457^{**}	0.0794^{**}	0.0722^{**}	0.0081
I	0.0266^{**}	0.0631^{**}	0.0656^{**}	-0.0002
J	0.0784^{**}	0.0993^{**}	0.0935^{**}	0.0037
K	0.0355^{**}	0.0701^{**}	0.0803^{**}	0.0003
L	0.0745^{**}	0.0828^{**}	0.0771^{**}	-0.0076
M	0.0247^{**}	0.0396^{**}	0.0379^{**}	0.0027
N	0.0344^{**}	0.0484^{**}	0.0449^{**}	0.0066^{*}
O	0.1052^{**}	0.0913^{**}	0.0997^{**}	-0.0012
P	0.0141	-0.0105	0.0012	-0.0101
Q	0.0549^{**}	0.0590^{**}	0.0530^{**}	-0.0101
R	0.0554^{**}	0.0437^{**}	0.0340^{**}	-0.0131
S	0.0226^{*}	0.0050	0.0099	0.0080
T	0.0363^{**}	0.0802^{**}	0.0730^{**}	-0.0102^{*}
U	0.0092	0.0419^{**}	0.0524^{**}	0.0073
V	-0.0023	0.0070	0.0342^{**}	0.0181^{*}
W	0.0387^{**}	0.0201^{**}	0.0085^{**}	-0.0032
X	0.0394^{**}	0.1212^{**}	0.1154^{**}	-0.0003
Y	0.0319^{**}	0.0483^{**}	0.0573^{**}	-0.0067
Z	0.0334^{**}	0.0630^{**}	0.0673^{**}	-0.0031
Metric mean	0.0438	0.0611	0.0631	0.0007

$^{*}\ p < 0.05$
$^{**}\ p < 0.01$

Table 8: Predictability of *Maintenance* from size-normalized metric values, measured by Kendall's τ_b. Values range between -1.0 and 1.0.

Size	USJ	GZP	MNK
A	0.0515^{**}	-0.0746^{**}	-0.0818^{**}
B	0.0605^{**}	-0.0376^{**}	-0.0690^{**}
C	0.0046	-0.0127	-0.0150
D	0.0683^{**}	-0.0487^{**}	-0.0518^{**}
E	0.0453^{**}	-0.0501^{**}	-0.0495^{**}
F	0.0411^{**}	-0.0926^{**}	-0.0799^{**}
G	0.0047	-0.0386^{**}	-0.0478^{**}
H	0.0355^{**}	-0.0608^{**}	-0.0570^{**}
I	0.0196^{**}	-0.0437^{**}	-0.0500^{**}
J	0.0701^{**}	-0.0949^{**}	-0.0760^{**}
K	0.0257^{**}	-0.0334^{**}	-0.0553^{**}
L	0.0732^{**}	-0.0706^{**}	-0.0643^{**}
M	0.0239^{**}	-0.0115^{**}	-0.0244^{**}
N	0.0334^{**}	-0.0442^{**}	-0.0326^{**}
O	0.0930^{**}	-0.0450^{**}	-0.0745^{**}
P	0.0143	0.0267^{**}	0.0010
Q	0.0420^{**}	-0.0586^{**}	-0.0539^{**}
R	0.0368^{**}	-0.0603^{**}	-0.0379^{**}
S	0.0184^{*}	0.0095	0.0063
T	0.0225^{**}	-0.0711^{**}	-0.0673^{**}
U	0.0048	0.0049	-0.0267^{**}
V	-0.0044	0.0223^{**}	0.0077
W	0.0374^{**}	-0.0310^{**}	-0.0173^{**}
X	0.0238^{*}	-0.1069^{**}	-0.1021^{**}
Y	0.0298^{**}	-0.0199^{**}	-0.0379^{**}
Z	0.0249^{**}	-0.0416^{**}	-0.0520^{**}
Metric mean	0.0346	-0.0417	-0.0465

$^{*}\ p < 0.05$
$^{**}\ p < 0.01$

The authors would like to thank Gal Lalouche. Research was supported by ISF grant 2020028.

6. REFERENCES

[1] F. E. Allen. Control flow analysis. *SIGPLAN Notices*, 5(7):1–19, July 1970.

[2] H. Barkmann, R. Lincke, and W. Lowe. Quantitative evaluation of software quality metrics in open-source projects. In *Advanced Information Networking and Applications Workshops, 2009. WAINA '09. International Conference on*, pages 1067–1072, May 2009.

[3] C. Böhm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, May 1966.

[4] E. W. Dijkstra. Letters to the editor: GOTO statement considered harmful. *Commun. ACM*, 11(3):147–148, Mar. 1968.

[5] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, 25(5):675–689, Sept. 1999.

[6] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson. Software defect prediction using static code metrics underestimates defect-proneness. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1–7, July 2010.

[7] D. Knuth. Structured programming with GOTO statements. In E. N. Yourdon, editor, *Classics in software engineering*, pages 257–321. Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[8] S. Markstrum. Staking claims: A history of programming language design claims and evidence: A positional work in progress. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 7:1–7:5, New York, NY, USA, 2010. ACM.

[9] T. J. McCabe. A complexity measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[10] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 284–292, New York, NY, USA, 2005. ACM.