Probabilistic Inference of Data Abstractions in Machine Code

Aditya Thakur and Mark Chapman
May 16, 2011

1 Introduction

Our problem and motivation

2 Overview

Our solution in general terms

Terminology

1. Classes (y): training set labels
2. Features (f): observed n-grams (n abstracted x86 instruction)
3. Counts (C_{fy}): number of times feature f appears with label y
4. Information Gain (I_{fy}): predictive power of feature f for label y
5. Train: choose best fingerprints by maximizing information gain
6. Test: find fingerprint matches in executables

Mutual Information

This guides our training set fingerprints to choose features which find data structure similarity rather than purely function similarity. The overall function we maximize is:

$$I = \sum_f \sum_y I_{fy}$$

Divided into features f and classes y:

$$I_{fy} = P_{fy} \log(\frac{P_{fy}}{P_f P_y})$$

Each probability P is approximated by the counts C in a training set:

$$P_{fy} = \frac{C_{fy}}{\sum_f \sum_y C_{fy}} = \frac{C_{fy}}{C}$$

$$P_f = \frac{\sum_y C_{fy}}{C} = \frac{C_f}{C}$$

$$P_y = \frac{\sum_f C_{fy}}{C} = \frac{C_y}{C}$$

This simplifies each information gain to three terms relying only on training set counts:

$$I_{fy} = \frac{C_{fy}}{C} \log(\frac{C_{fy}}{C_y} \frac{C}{C_f})$$

The first term avoids overfitting to extremely rare n-grams by weighting n-grams by their observed frequency. The second term weights highly n-grams which are common within a class. The final term lowers weight of n-grams which are common in many classes.

Fingerprinting

For each class y, we rank n-gram features f which appear in the class according to information gain:

$$f_1 > f_2 > \cdots > f_v \Leftrightarrow I_{f_1 y} > I_{f_2 y} > \cdots > I_{f_v y} > 0$$

For each block of machine code fingerprinted, store the top k features in the block as the fingerprint. We chose functions as a convenient level of granulation.

3 Related work

Related work

Early work that finds static library code using hash collisions [1].

Introduction to pointers, arrays, and recursive structures in binary [2].

Closest to our technique: pick fingerprints, calculate similarities [3].

Uses similar code region normalization; relies on approximate hashing instead of fingerprints for similarity [4]

Recent survey covering variety of methods to find similar code segments [5]

4 Current status

Current status of our project

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Experiment Set I

Controlled Training and Testing (distance < 0.2)						
	Training Set	Test Set	exe's/func's/loc's	TP	FP	FN
1	avltree, stackli, tree	testavl, teststkl, testtree	3 / 84 / 1629	18	0	9
2	list, tree	sll-reverse	1 / 26 / 516	0	0	7
3	list, tree, sll-reverse	libraries: abl, dsaac, sglib	83 / 2865 / 95451	95	2	847
4	dll and all from 3	same libraries as 3	83 / 2865 / 95451	169	2	988

Experiment 3											
distance:	< 0.1	< 0.2	< 0.3	< 0.4	< 0.5	< 0.6	< 0.7	< 0.8	< 0.9	< 1.0	Total
sll:	43	48	95	98	124	150	265	322	484	527	609
tree:	35	47	98	99	107	117	179	206	247	264	333
FP:	0	2	182	182	191	321	605	705	743	759	1708
Precision:	1.00	0.98	0.51	0.52	0.55	0.45	0.42	0.43	0.50	0.51	0.36
Recall:	0.08	0.10	0.20	0.21	0.25	0.28	0.49	0.56	0.78	0.84	1.00

Experiment 4											
distance:	< 0.1	< 0.2	< 0.3	< 0.4	< 0.5	< 0.6	< 0.7	< 0.8	< 0.9	< 1.0	Total
dll:	43	70	184	184	185	192	193	202	212	215	215
sll:	47	52	71	74	94	124	188	295	430	527	609
tree:	35	47	92	93	101	107	147	190	236	261	333
FP:	0	2	189	189	211	330	608	701	741	759	1708
Precision:	1.00	0.99	0.65	0.65	0.64	0.56	0.46	0.49	0.54	0.57	0.40
Recall:	0.11	0.15	0.30	0.30	0.33	0.37	0.46	0.59	0.76	0.87	1.00

Experiment Set II

Finding Data Structures in a Large Library (distance < 0.2)				
	Training Set	Test Set	so's/func's/loc's	possible matches
1	dll, list, tree, sll-reverse	libglib-2.0	1 / 1245 / 81952	15

5 Future work

Next steps in our project

References

[1] M. V. Emmerik, "Identifying library functions in executable files using patterns," in *Australian Software Engineering Conference (ASWEC '98)*, pp. 90–97, IEEE Computer Society, 1998.

[2] A. Mycroft, "Type-based decompilation (or program reconstruction via type reconstruction)," in *European Symposium on Programming (ESOP '99)* (S. D. Swierstra, ed.), vol. 1576 of *Lecture Notes in Computer Science*, pp. 208–223, Springer, 1999.

[3] R. Smith and S. Horwitz, "Detecting and measuring similarity in code clones," in *International Workshop on Software Clones (IWSC 2009)*, Computer, pp. 28–34, 2009.

[4] A. Sæbjœrnsen, J. Willcock, T. Panas, D. J. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *International Symposium on Software Testing and Analysis (ISSTA 2009)* (G. Rothermel and L. K. Dillon, eds.), pp. 117–128, ACM, 2009.

[5] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.