Simpler Software Analytics: When? When not?

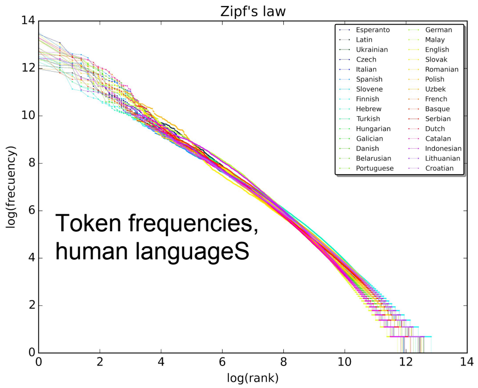
Wei Fu

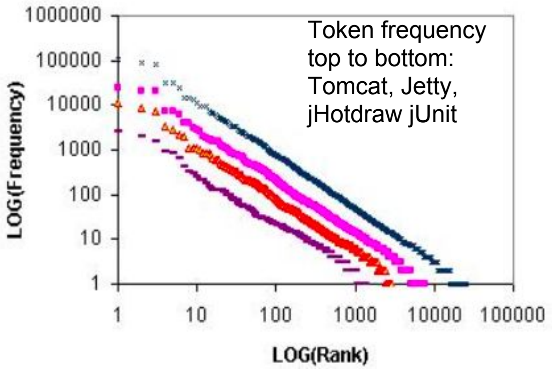
From Last Exam

Why study simplicity? *cost, speed*

When this won't work? *ε-Dominance*

What's the difference between SE/general data mining? *under-exploited simplicities*





* Hindle, Abram, et al. "On the naturalness of software." Software Engineering (ICSE), 2012 34th International Conference on. IEEE, 2012.

My Thesis:

Software analytics **should** be easier.
Software analytics **can** be easier.

But it can be **very hard** to show it can be easier.

And, sometimes, it can be **too easy**.

Future work:

When to be simpler.

My Thesis

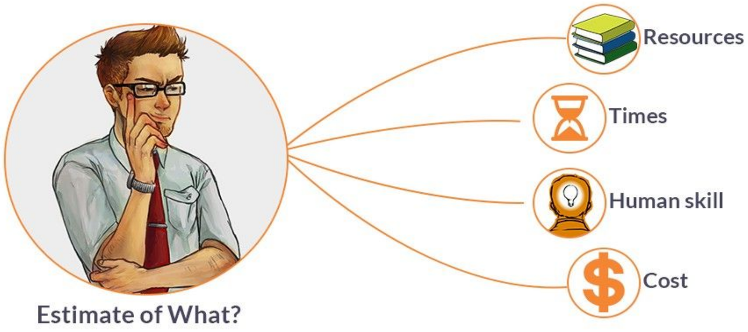
Software analytics **should** be easier.

Software analytics **can** be easier. [Fu et al. IST 2016]

But it can be **very hard** to show it can be easier. [Fu et al. FSE 2017 A]

And, sometimes, it can be **too easy**. [Fu et al. FSE 2017 B]

Software Analytics

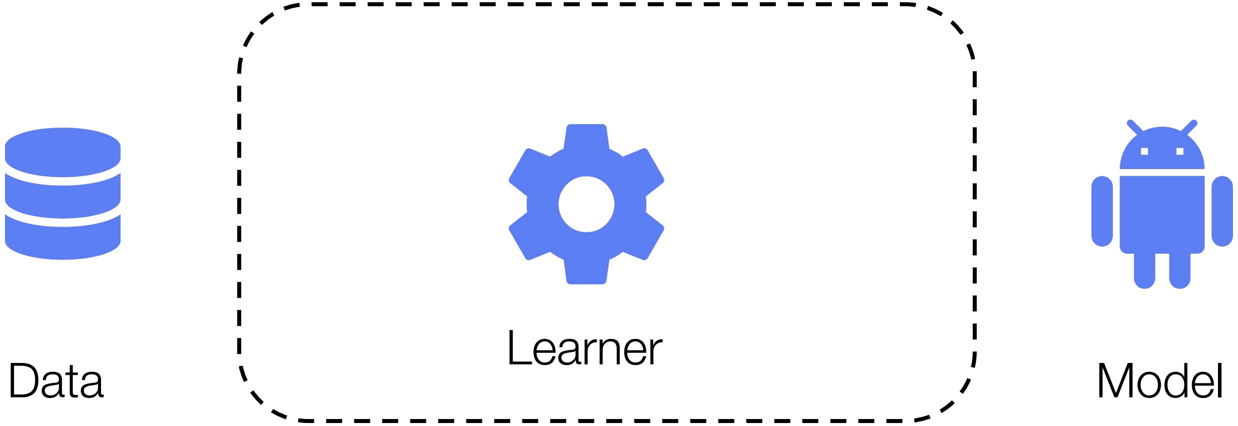


Effort estimation

Software Defect Prediction

Many Others…..

A Typical Software Analytics Framework



Training process

2007 TSE

2 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 33, NO. 1, JANUARY 2007

Data Mining Static Code Attributes to Learn Defect Predictors

Tim Menzies, *Member*, *IEEE*, Jeremy Greenwald, and Art Frank

Abstract—The value of using static code attributes to learn defect predictors has been widely debated. Prior work has explored issues like the merits of "McCabes versus Halstead versus lines of code counts" for generating defect predictors. We show here that such debates are irrelevant since *how* the attributes are used to build predictors is much more important than *which* particular attributes are used. Also, contrary to prior pessimism, we show that such defect predictors are demonstrably useful and, on the data studied here, yield predictors with a mean probability of detection of 71 percent and mean false alarms rates of 25 percent. These predictors would be useful for prioritizing a resource-bound exploration of code that has yet to be inspected.

Index Terms—Data mining detect prediction, McCabe, Halstead, artifical intelligence, empirical, naive Bayes.

Decision Tree, Naive Bayes

2007 TSE

Data Mining Static Code Attributes to Learn Defect Predictors

Tim Menzies, *Member*, *IEEE*, Jeremy Greenwald, and Art Frank

Abstract—The value of using static code attributes to learn defect predictors has been widely debated. Prior work has explored issues like the merits of "McCabes versus Halstead versus lines of code counts" for generating defect predictors. We show here that such debates are irrelevant since *how* the attributes are used to build predictors is much more important than *which* particular attributes are used. Also, contrary to prior pessimism, we show that such defect predictors are demonstrably useful and, on the data studied here, yield predictors with a mean probability of detection of 71 percent and mean false alarms rates of 25 percent. These predictors would be useful for prioritizing a resource-bound exploration of code that has yet to be inspected.

Index Terms—Data mining detect prediction, McCabe, Halstead, artifical intelligence, empirical, naive Bayes.

Decision Tree, Naive Bayes

2016 ICSE

2016 IEEE/ACM 38th IEEE International Conference on Software Engineering

Automatically Learning Semantic Features for Defect Prediction

Song Wang,Taiyue Liu and Lin Tan
Electrical and Computer Engineering, University of Waterloo, Canada
{song.wang, t67liu, lintan}@uwaterloo.ca

ABSTRACT

Software defect prediction, which predicts defective code regions, can help developers find bugs and prioritize their testing efforts. To build accurate prediction models, previous studies focus on manually designing features that encode the characteristics of programs and exploring different machine learning algorithms. Existing traditional features often fail to capture the semantic differences of programs, and such a capability is needed for building accurate prediction models.

To bridge the gap between programs' semantics and defect prediction features, this paper proposes to leverage a powerful representation-learning algorithm, deep learning, to learn semantic representation of programs automatically from source code. Specifically, we leverage Deep Belief Network (DBN) to automatically learn semantic features from token vectors extracted from programs' Abstract Syntax Trees (ASTs).

machine learning algorithms. Researchers have manually designed many features to distinguish defective files from non-defective files, e.g., Halstead features [10] based on operator and operand counts, McCabe features [31] based on dependencies, CK features [5] based on function and inheritance counts, etc., MOOD features [11] based on polymorphism factor, coupling factor, etc., code change features [18] include number of lines of code added, removed, etc., and other object-oriented features [7]. Meanwhile, many machine learning algorithms have been adopted for software defect prediction, including Support Vector Machine (SVM), Naive Bayes (NB), Decision Tree (DT), Neural Network (NN), and Dictionary Learning [20].

Programs have well-defined *syntax*, which can be represented by Abstract Syntax Trees (ASTs) [15] and have been successfully used to capture programming patterns [44, 46]. In addition, programs have *semantics*, which is hidden

Deep Learning

Simpler or more complex software analytics?

My Thesis

Software analytics **should** be easier.

Software analytics **can** be easier. [Fu et al, IST 2016]

But it can be **very hard** to show it can be easier. [Fu et al, FSE 2017 A]

And, sometimes, it can be **too easy**. [Fu et al, FSE 2017 B]

Why Easier: Cost & Speed

Dr. Mark Harman@UCL

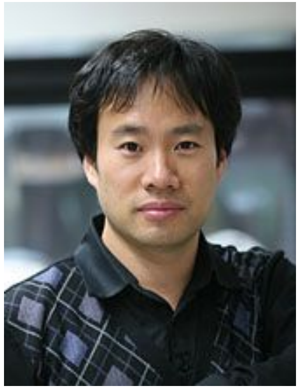
FSE'13: Wang et al[Wang13]

Wait **Years** of CPU time

Dr. Tien N. Nguyen@UTDallas

ASE'15: Lam et al[Lam15]

Wait **Weeks** of CPU time



Dr. Sung Kim@HKUST

FSE'16: Gu et al[Gu16]

Wait **10 Days** of GPU time

Dr. Tim Menzies@NCSU

FSE'17: Fu et al[Fu17]

Wait **10 minutes** of CPU time

Why Easier: Cost & Speed

Dr. Devanbu@UC Davis

FSE'17: Hellendoorn et al[Hellendorrn17]

Simpler, faster methods, complex DL is not always the best.

Are Deep Neural Networks the Best Choice for Modeling Source Code?

Vincent J. Hellendoorn
Computer Science Dept., UC Davis
Davis, CA, USA 95616
vhellendoorn@ucdavis.edu

Premkumar Devanbu
Computer Science Dept., UC Davis
Davis, CA, USA 95616
ptdevanbu@ucdavis.edu

ABSTRACT

Current statistical language modeling techniques, including deep-learning based models, have proven to be quite effective for source code. We argue here that the special properties of source code can be exploited for further improvements. In this work, we enhance established language modeling approaches to handle the special challenges of modeling source code, such as: frequent changes, larger, changing vocabularies, deeply nested scopes, etc. We present a fast, nested language modeling toolkit specifically designed for software, with the ability to add & remove text, and mix & swap out many models. Specifically, we improve upon prior cache-modeling work and present a model with a much more expansive, multi-level notion of locality that we show to be well-suited for modeling software. We present results on varying corpora in comparison with traditional *N*-gram, as well as RNN, and LSTM deep-learning language models, and release all our source code for public use. Our evaluations suggest that carefully adapting *N*-gram models for source code can yield performance that surpasses even RNN and LSTM based deep-learning models.

Statistical models from NLP, estimated over the large volumes of code available in GitHub, have led to a wide range of applications in software engineering. High-performance language models are widely used to improve performance on NLP-related tasks, such as translation, speech-recognition, and query completion; similarly, better language models for source code are known to improve performance in tasks such as code completion [15]. Developing models that can address (and exploit) the special properties of source code is central to this enterprise.

Language models for NLP have been developed over decades, and are highly refined; however, many of the design decisions baked-into modern NLP language models are finely-wrought to exploit properties of natural language corpora. These properties aren't always relevant to source code, so that adapting NLP models to the special features of source code can be helpful. We discuss 3 important issues and their modeling implications in detail below.

Unlimited Vocabulary Code and NL can both have an unbounded vocabulary; however, in NL corpora, the vocabulary usually saturates quickly: when scanning through a large NL corpus, pretty soon, one rarely encounters new words. New proper nouns (people

Why Easier: Cost

Local hardware:

- ***AlphaGo***: 1920 CPUs and 280 GPUs*, $3000 electric bill per game

AlphaGo

* https://en.wikipedia.org/wiki/AlphaGo

My Thesis

Software analytics **should** be easier.

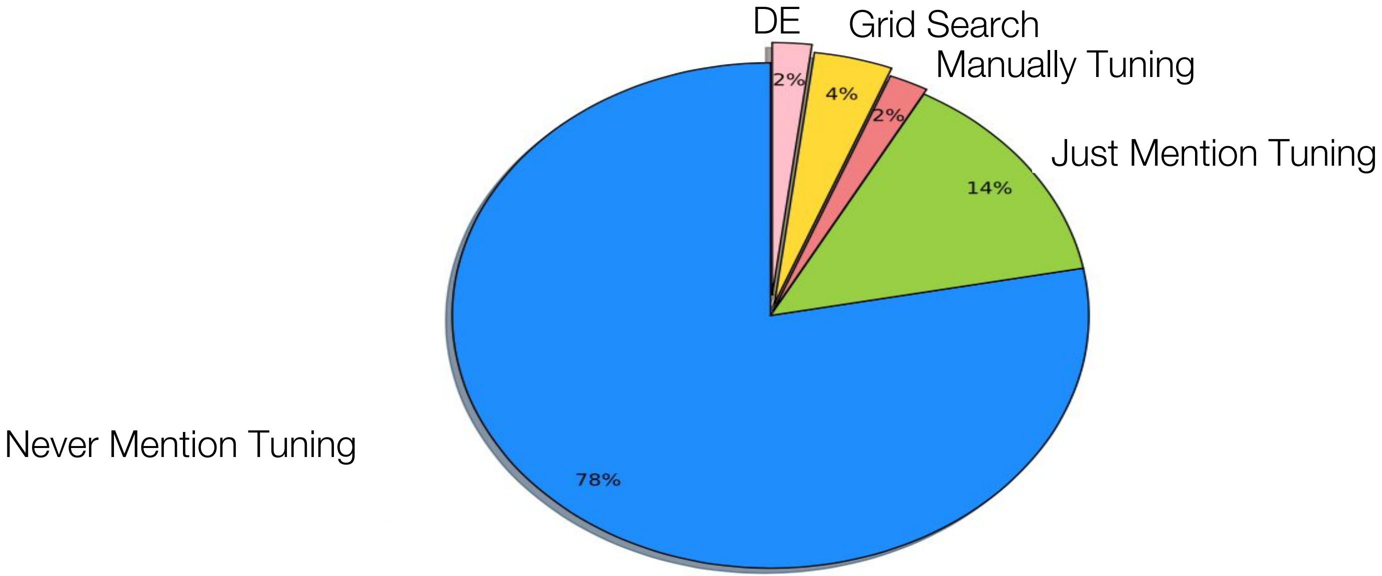
Software analytics **can** be easier. [Fu et al, IST 2016]

But it can be **very hard** to show it can be easier. [Fu et al, FSE 2017 A]

And, sometimes, it can be **too easy**. [Fu et al, FSE 2017 B]

Fu, Wei, Tim Menzies, and Xipeng Shen. "**Tuning for software analytics: Is it really necessary?.**" *Information and Software Technology* 76 (2016): 135-146.

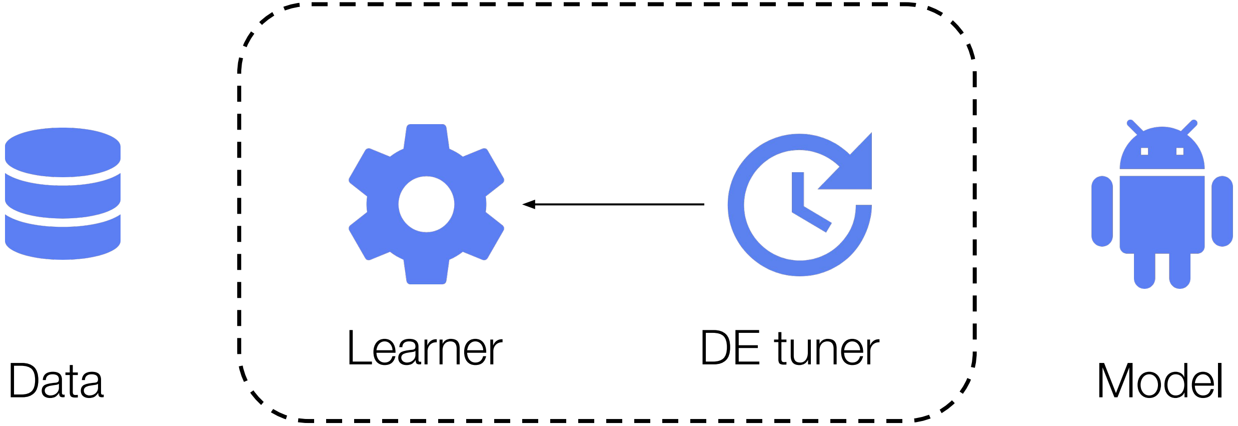
Tuning is Ignored in SE



Literature Review On Defect Prediction*

* Fu, Wei, Vivek Nair, and Tim Menzies. "Why is Differential Evolution Better than Grid Search for Tuning Defect Predictors?." arXiv preprint arXiv:1609.02613 (2016).

Tuning Defect Predictors



Training process

Differential Evolution

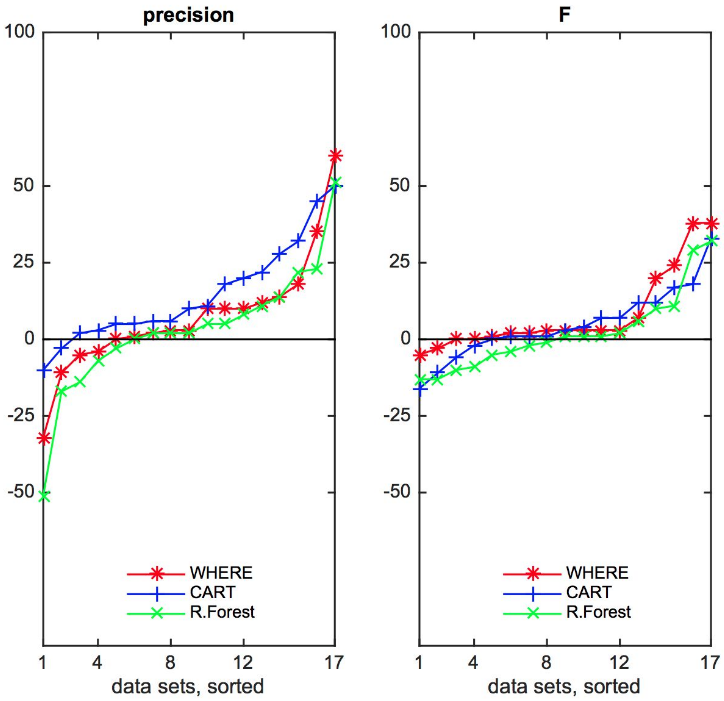
Population = Pick N options at random # e.g. N =10

M times repeat : # e.g. M = 5

for Parent in Population:

- Select a, b, c = three other items in population.
- Candidate = a + f*(b-c) # ish
- if Candidate “better”, replace Parent.

Tuning Defect Predictors



Time Cost of Tuning Defect Predictors

Datasets	Tuned_CART		Naive_CART		Tuned_RanFst		Naive_RanFst	
	precision	F	precision	F	precision	F	precision	F
antV0	5.08	3.52	0.08	0.08	9.78	9.89	0.20	0.17
antV1	6.52	6.18	0.12	0.12	14.13	13.39	0.25	0.25
antV2	9.00	8.79	0.24	0.18	16.75	27.56	0.44	0.36
camelV0	12.68	17.00	0.24	0.28	28.49	22.52	0.34	0.41
camelV1	17.13	31.92	0.27	0.28	33.96	37.00	0.77	0.85
ivy	4.26	4.72	0.07	0.08	8.89	10.39	0.19	0.21
jeditV0	8.69	7.9	0.11	0.10	18.40	14.32	0.32	0.37
jeditV1	9.05	8.13	0.12	0.10	17.93	17.42	0.36	0.34
jeditV2	7.90	10.34	0.14	0.15	27.34	20.20	0.38	0.40
log4j	2.60	2.92	0.06	0.08	9.69	7.67	0.15	0.17
lucene	6.07	6.89	0.10	0.12	9.77	13.06	0.25	0.35
poiV0	7.42	7.80	0.09	0.10	25.86	19.29	0.28	0.32
poiV1	9.31	7.62	0.13	0.14	12.67	27.23	0.29	0.36
synapse	3.88	4.87	0.07	0.08	8.13	13.29	0.19	0.17
velocity	4.27	5.51	0.07	0.10	15.18	11.58	0.21	0.27
xercesV0	9.17	7.47	0.10	0.11	14.17	17.31	0.22	0.28
xercesV1	10.47	11.07	0.16	0.19	18.27	25.27	0.40	0.46

x62

x45

My Thesis

Software analytics **should** be easier.

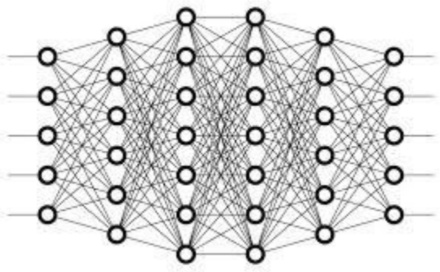
Software analytics **can** be easier. [Fu et al, IST 2016]

But it can be **very hard** to show it can be easier. [Fu et al, FSE 2017 A]

And, sometimes, it can be **too easy**. [Fu et al, FSE 2017 B]

Our Objective

Data



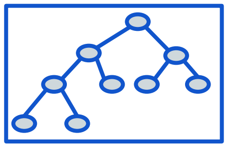
Complex Method (e.g., Deep Learning)

Model

Training process

Our Objective

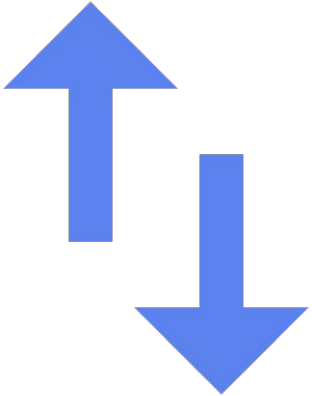
Data



Simple method

Model

Training process



Wei Fu, and Tim Menzies. "**Easy over hard: a case study on deep learning.**" In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 49-60. ACM, 2017.

Deep Learning in SE

- From 2015 to 2017, 11 DL paper in SE
- 4 Papers mentioned training cost
- None compares DL costs with competitor methods

How Hard Can It Be ?

- Baseline methods are not well described
- No Data, No DL Code
- Did not report DL costs

What I Got

Predicting Semantically Linkable Knowledge in Developer Online Forums via Convolutional Neural Network

Bowen Xu[1] *, Deheng Ye[2] *, Zhenchang Xing[2], Xin Xia[1] †, Guibin Chen[2], Shanping Li[1]
[1]College of Computer Science and Technology, Zhejiang University, China
[2]School of Computer Science and Engineering, Nanyang Technological University, Singapore
max_xbw@zju.edu.cn, ye0014ng@e.ntu.edu.sg, zcxing@ntu.edu.sg,
xxia@zju.edu.cn, gbchen@ntu.edu.sg, shan@zju.edu.cn

ABSTRACT

Consider a question and its answers in Stack Overflow as a knowledge unit. Knowledge units often contain semantically relevant knowledge, and thus linkable for different purposes, such as duplicate questions, directly linkable for problem solving, indirectly linkable for related information. Recognising different classes of linkable knowledge would support more targeted information needs when users search or explore the knowledge base. Existing methods focus on binary relatedness (i.e., related or not), and are not robust to recognize different classes of semantic relatedness when linkable knowledge units share few words in common (i.e., have lexical gap). In this paper, we formulate the problem of predicting semantically linkable knowledge units as a multiclass classification problem, and solve the problem using deep learning techniques. To overcome the lexical gap issue, we adopt neural language model (word embeddings) and convolutional neural network (CNN) to capture word- and document-level semantics of knowledge units. Instead of using human-engineered classifier features which are hard to design for informal user-generated content, we exploit large amounts of different types of user-created knowledge-unit links to train the CNN to learn the most informative word-level and document-level features for the multiclass classification task. Our evaluation shows that our deep-learning based approach significantly and consistently outperforms traditional methods using traditional word representations and human-engineered classifier features.

Keywords

Link prediction, Semantic relatedness, Multiclass classification, Deep learning, Mining software repositories

1. INTRODUCTION

In Stack Overflow, computer programming knowledge has been shared through millions of questions and answers. We consider a Stack Overflow question with its entire set of answers as a *knowledge unit* regarding some programming-specific issues. The knowledge contained in one unit is likely to be related to knowledge in other units. When asking a question or providing an answer in Stack Overflow, users reference existing questions and answers that contain relevant knowledge by URL sharing [46], which is strongly encouraged by Stack Overflow [2]. Through URL sharing, a network of *linkable knowledge units* has been formed over time [46].

Unlike linked pages on Wikipedia that follows the underlying knowledge structure, questions and answers are specific to individual's programming issues, and URL sharing in Q&As is opportunistic, because it is based on the community awareness of the presence of relevant questions and answers. A recent study by Ye et al. [46] shows that the structure of the knowledge network that URL sharing activities create is scale free. A scale free network follows a power law degree distribution, which can be explained using preferential attachment theory [4], i.e., "the rich get richer". On

ASE'16

Given two questions from stack overflow, are they ***duplicate***, ***direct link, indirect link*** or ***isolated***?

Comparison

ASE’16(Xu et al.)

Baseline: SVM

Proposed: CNN

FSE’17(Fu et al.)

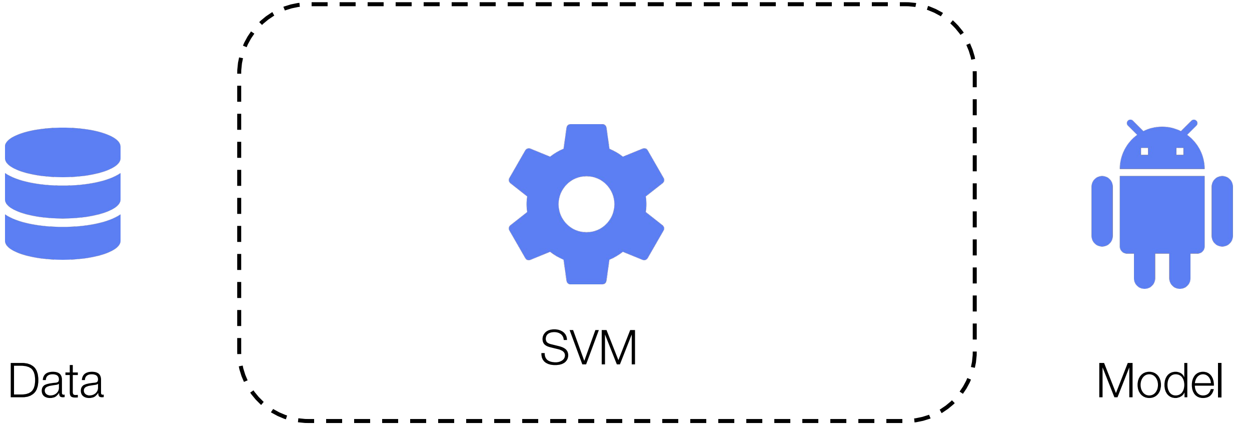
Baseline: CNN

Proposed: SVM+DE

What I Did Over One Month

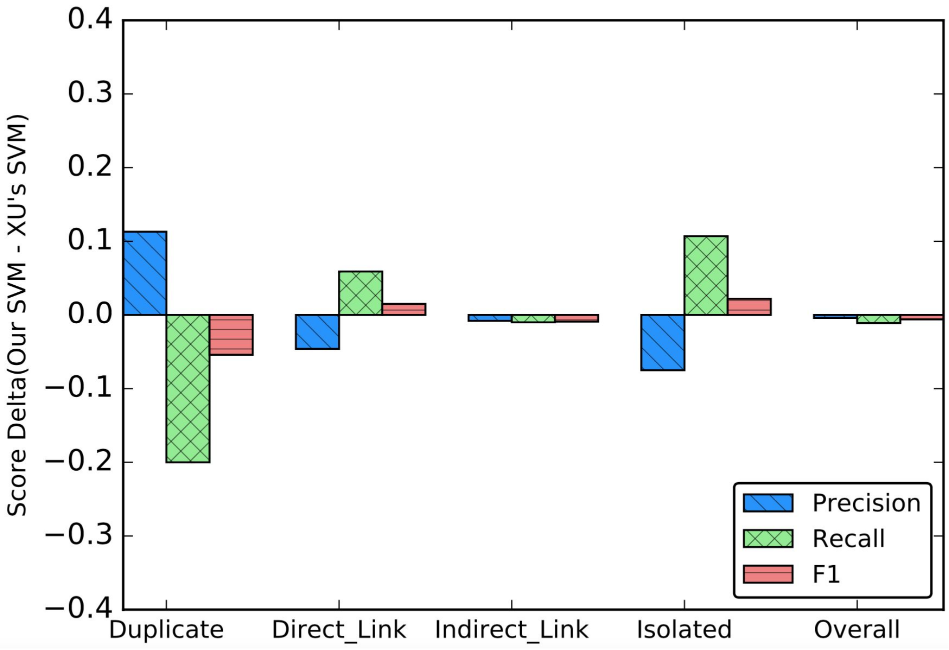
- Collect data from Stack Overflow (60 GB)
- Pre-process data
- Follow Xu et al, replicate their experiment

Xu et al. Baseline Method



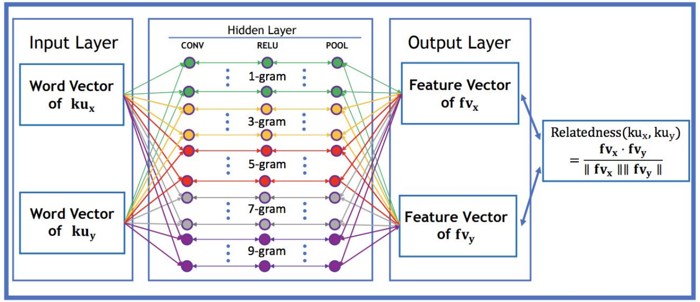
Training process

Successfully Reproduce Xu's Baseline

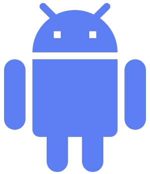


Xu et al's Complex Method: CNN

Data



Convolutional Neural Networks



Model

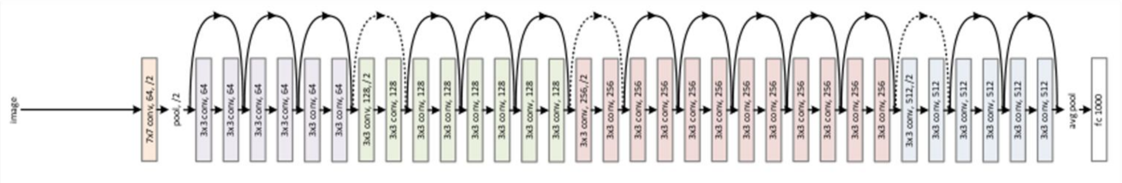
Training process

Typical CNN Architectures

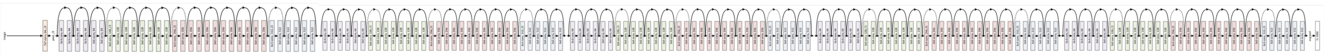
AlexNet-8

VGG-16

Resnet-34



Resnet-152



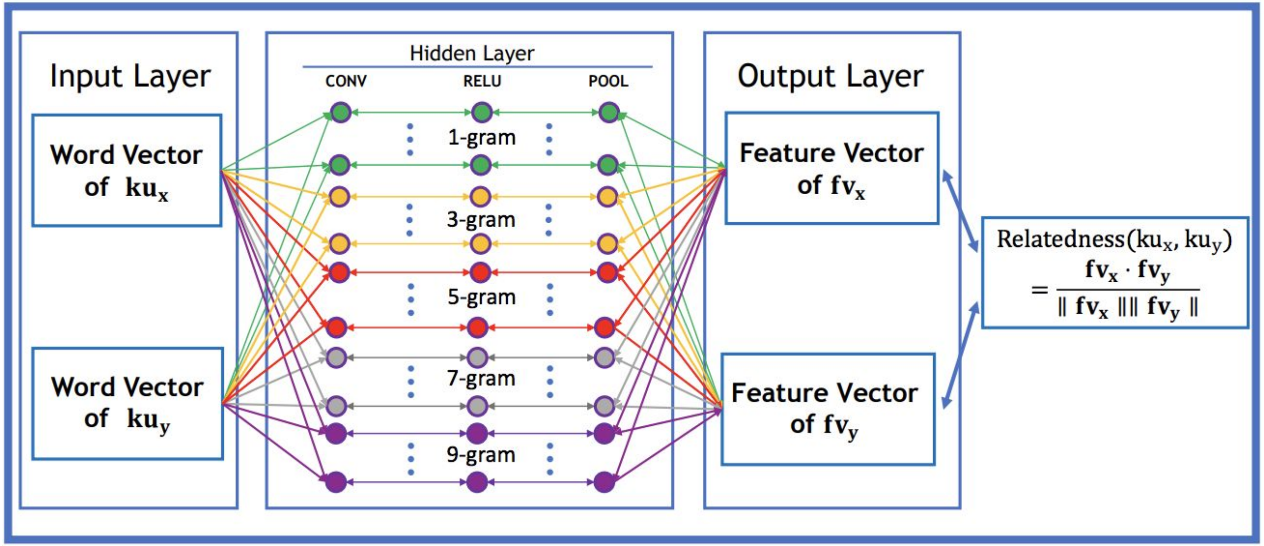
* http://sqlml.azurewebsites.net/2017/09/12/convolutional-neural-network/

Xu et al's CNN Architecture

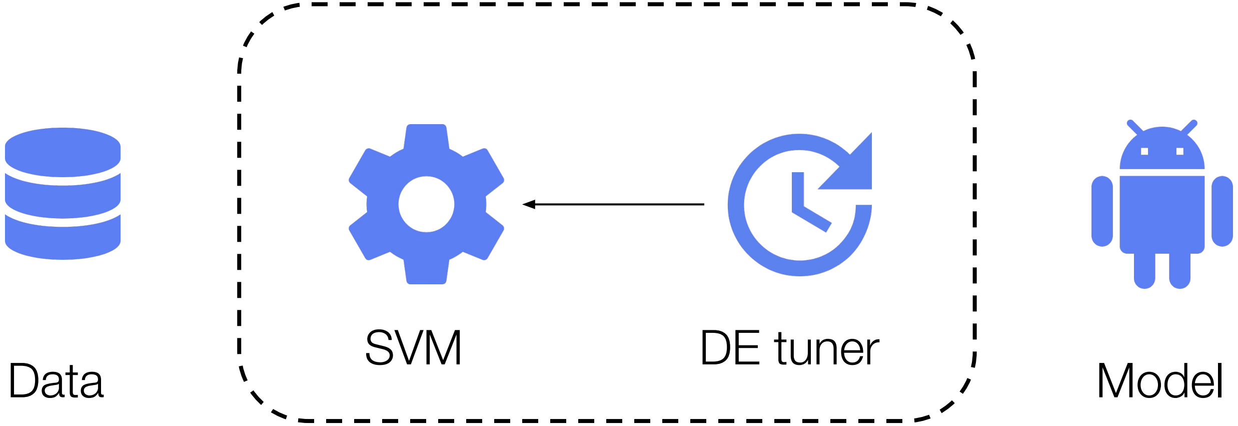
CONV: a dot product

RELU: max(0,x)

POOL: downsampling

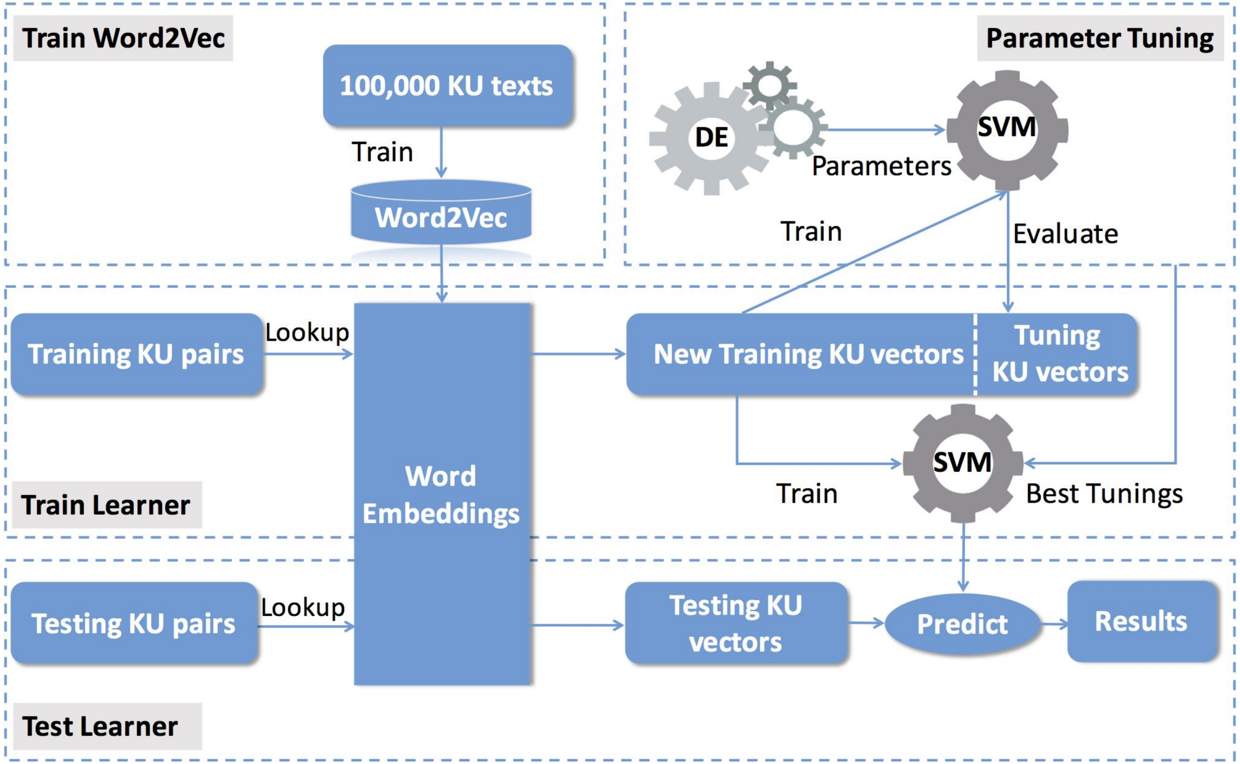


Simple Method: Tuning SVM With DE

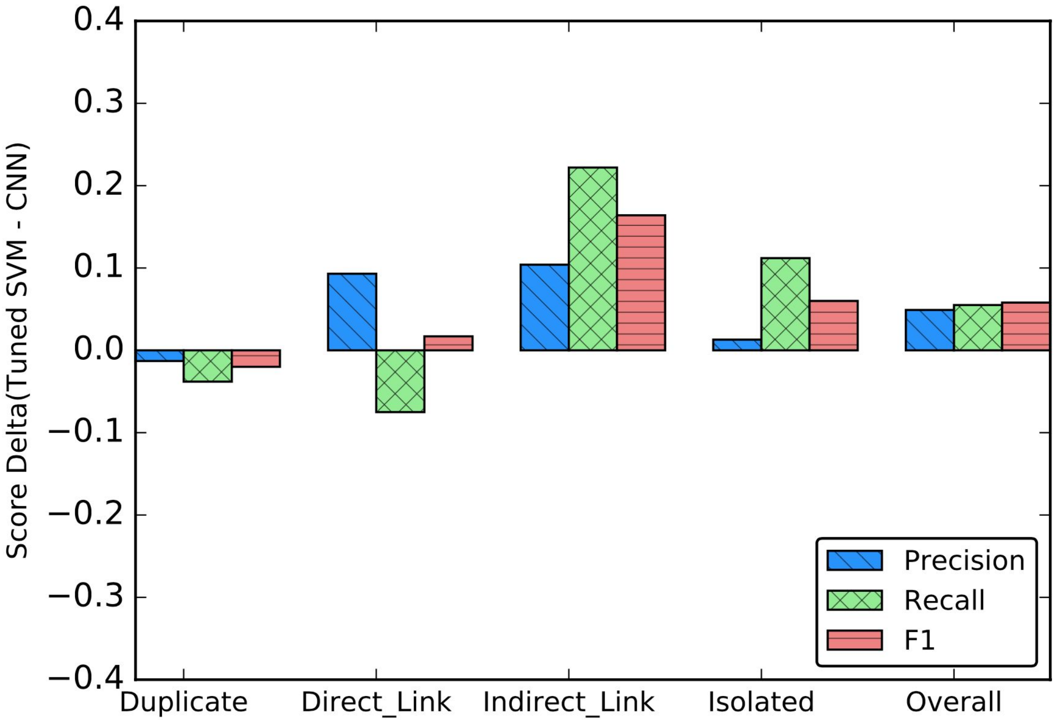


Training process

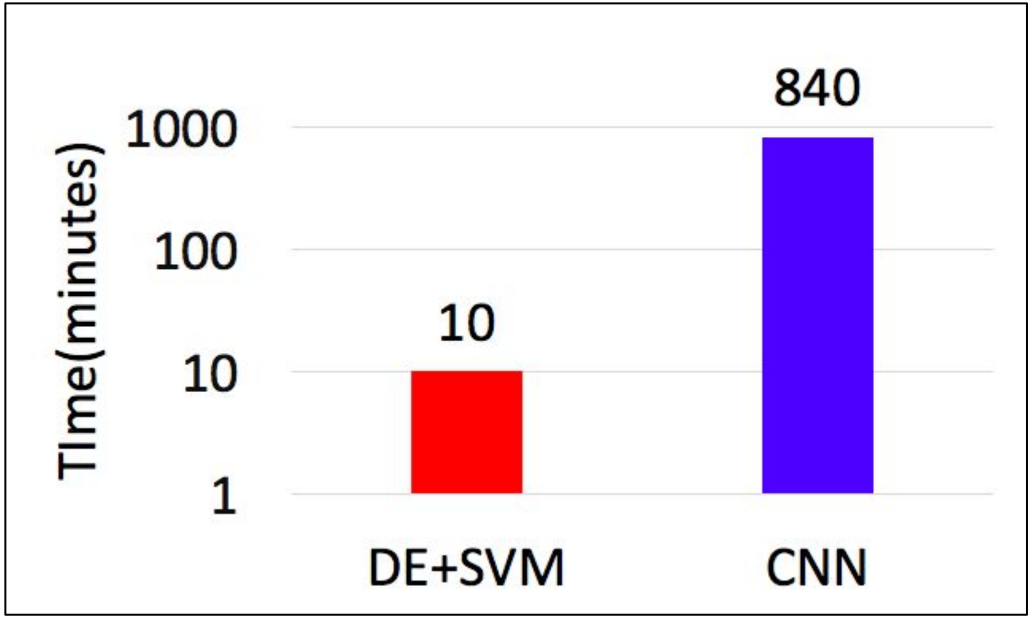
More Details



Easy Over Hard: Simplicity = Better results



Easy Over Hard: Less Runtime



My Thesis

Software analytics **should** be easier.

Software analytics **can** be easier. [Fu et al, IST 2016]

But it can be **very hard** to show it can be easier. [Fu et al, FSE 2017 A]

And, sometimes, it can be **too easy**. [Fu et al, FSE 2017 B]

Wei Fu, and Tim Menzies. "**Revisiting unsupervised learning for defect prediction.**" In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 72-83. ACM, 2017.

Effort-Aware Just-in-Time Defect Prediction: Simple Unsupervised Models Could Be Better Than Supervised Models

Yibiao Yang[1], Yuming Zhou[1*], Jinping Liu[1], Yangyang Zhao[1], Hongmin Lu[1], Lei Xu[1], Baowen Xu[1], and Hareton Leung[2]

[1] Department of Computer Science and Technology, Nanjing University, China

[2] Department of Computing, Hong Kong Polytechnic University, Hong Kong, China

ABSTRACT

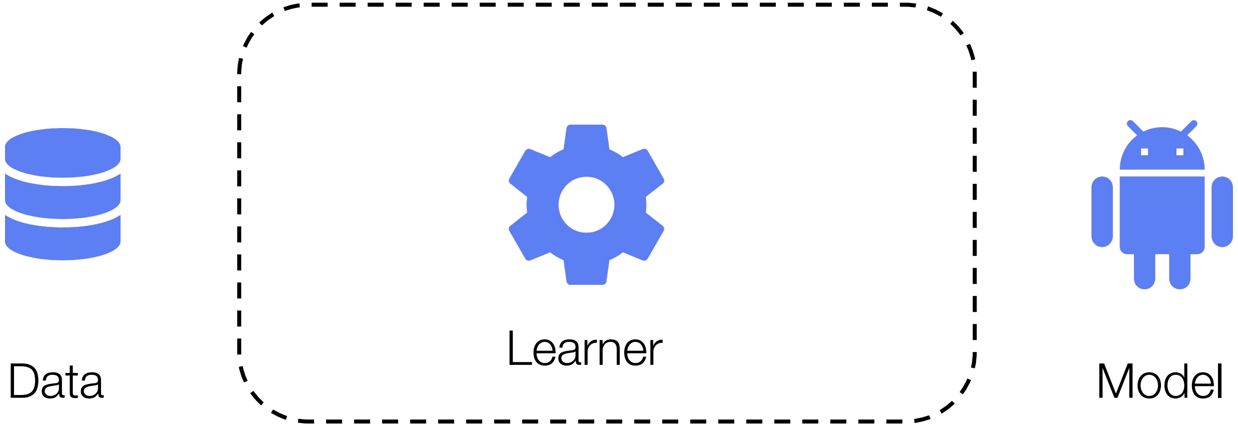
Unsupervised models do not require the defect data to build the prediction models and hence incur a low building cost and gain a wide application range. Consequently, it would be more desirable for practitioners to apply unsupervised models in effort-aware just-in-time (JIT) defect prediction if they can predict defect-inducing changes well. However, little is currently known on their prediction effectiveness in this context. We aim to investigate the predictive power of simple unsupervised models in effort-aware JIT defect prediction, especially compared with the state-of-the-art supervised models in the recent literature. We first use the most commonly used change metrics to build simple unsupervised models. Then, we compare these unsupervised models with the state-of-the-art supervised models under cross-validation, time-wise-cross-validation, and across-project prediction settings to determine whether they are of practical value. The experimental results, from open-source software systems, show that many simple unsupervised models perform better than the state-of-the-art supervised models in effort-aware JIT defect prediction.

consecutive commits in a given period of time) that introduce one or several defects into the source code in a software system [37]. Compared with traditional defect prediction at module (e.g. package, file, or class) level, JIT defect prediction is a fine granularity defect prediction. As stated by Kamei et al. [13], it allows developers to inspect an order of magnitude smaller number of SLOC (source lines of code) to find latent defects. This could provide large savings in effort over traditional coarser granularity defect predictions. In particular, JIT defect prediction can be performed at check-in time [13]. This allows developers to inspect the code changes for finding the latent defects when the change details are still fresh in their minds. As a result, it is possible to find the latent defects faster. Furthermore, compared with conventional non-effort-aware defect prediction, effort-aware JIT defect prediction takes into account the effort required to inspect the modified code for a change [13]. Consequently, effort-aware JIT defect prediction would be more practical for practitioners, as it enables them to find more latent defects per unit code inspection effort. Currently, there is a significant strand of interest in developing effective effort-aware JIT defect prediction models [7, 13].

FSE'16

A Typical Software Analytics Framework

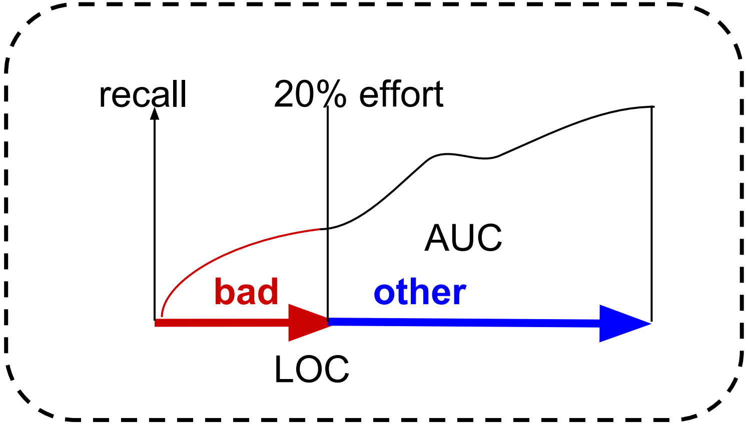
Supervised



Training process

Yang et al: Unsupervised Framework

Testing Data



Results

Testing Process

[Koru 2010]Koru, Gunes, et al. "Testing the theory of relative defect proneness for closed-source software." *Empirical Software Engineering* 15.6 (2010): 577-598.

More Details Over Here

Build 12 unsupervised models, on testing data:

10% effort

20% effort

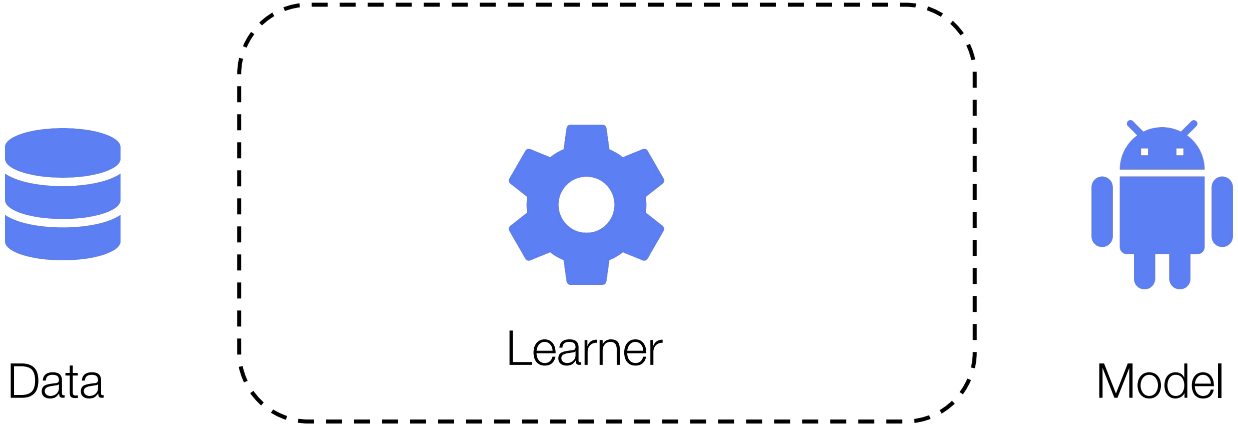
NF	NS	LT	FIX	ND	NDEV	EXP	REXP	SEXP	NUC	AGE	Entropy	LOC	Label
0	3	**11**	1	1	23	2	12	4	2	8	0.3	32	?
4	3	**24**	0	5	2	3	13	3	1	6	0.4	42	?
9	1	**89**	0	3	5	5	3	2	3	4	0.6	18	?
1	3	**34**	0	3	6	7	9	3	5	3	0.2	103	?
0	0	**537**	0	2	8	2	22	9	7	12	0.3	20	?
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Predicted as “Defective”

Our comments

- Reported averaged results across all projects
- How to apply 12 unsupervised learners in practice

A Typical Software Analytics Framework



Training process

OneWay

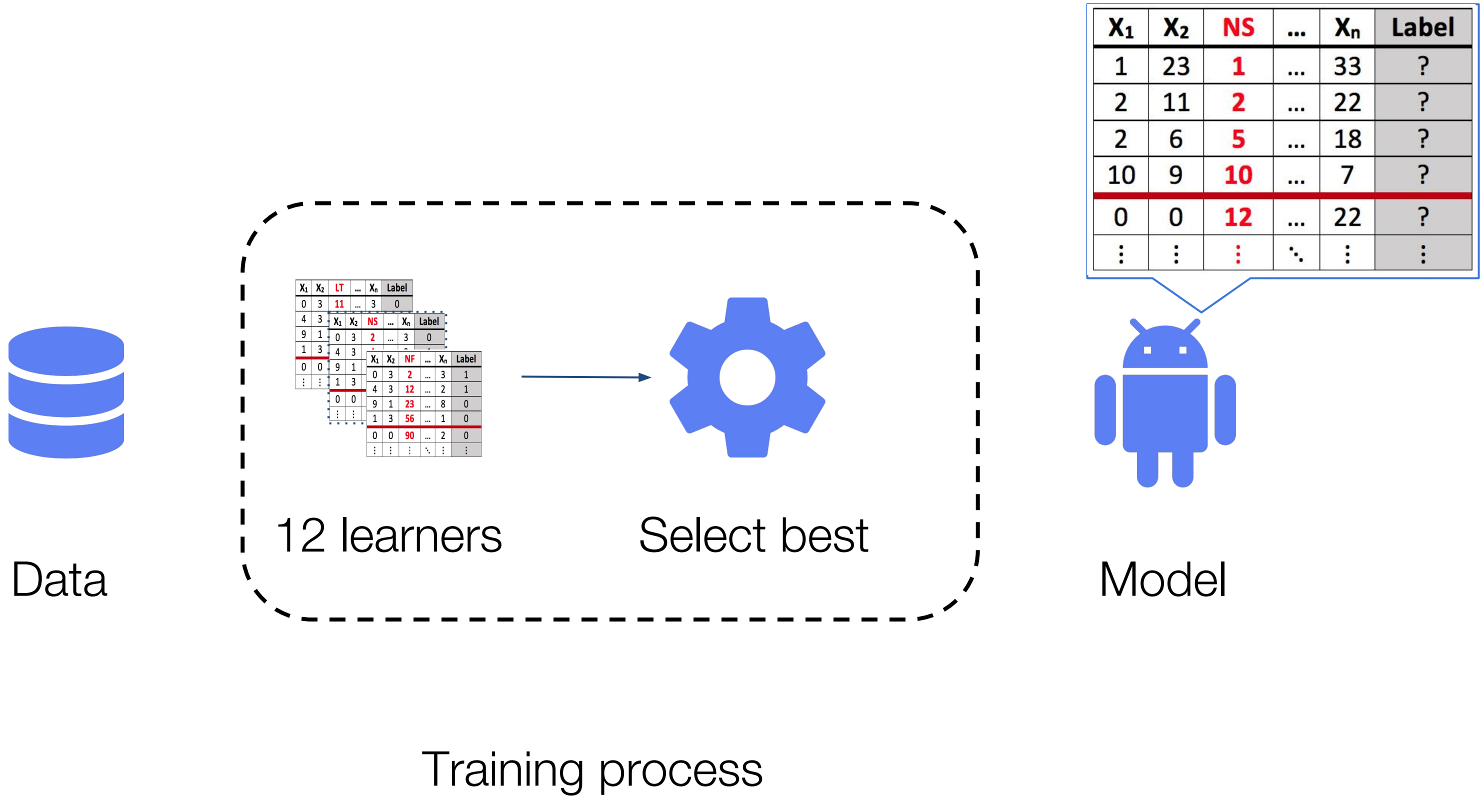
OneWay is not “the Way”

OneWay:

“The alternative way, maybe not the best way!”

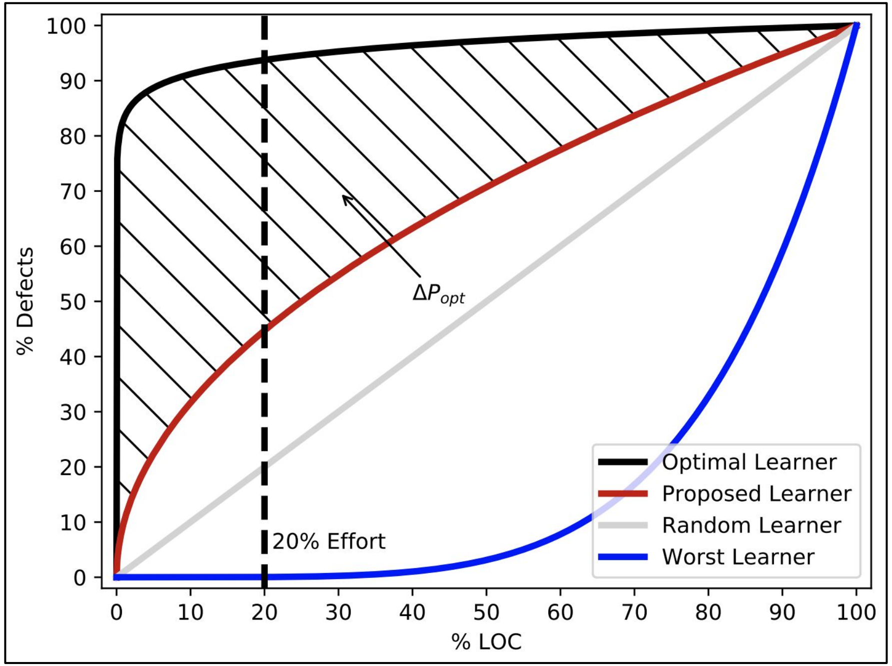
--Wei

OneWay Framework



Performance Measure

- Recall
- Popt
- F1
- Precision



$$P_{opt}(m) = 1 - \frac{S(optimal) - S(m)}{S(optimal) - S(worst)}$$

(Larger = Better)

Our Result Format

Bugzilla

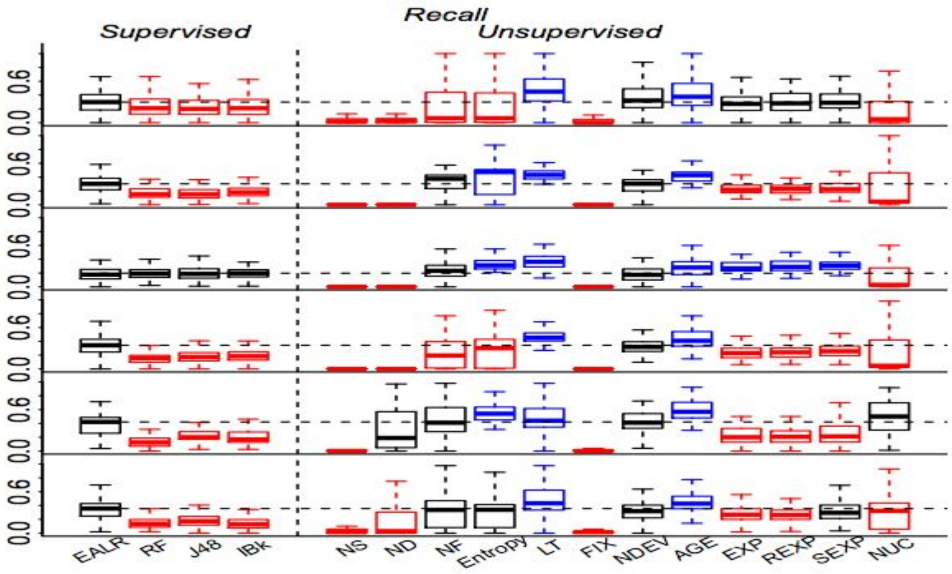
Platform

Mozilla

JDT

Columba

PostgreSQL



Blue: Better

Black: Similar

Red: Worse

Recall

Report results on a project-by-project basis.

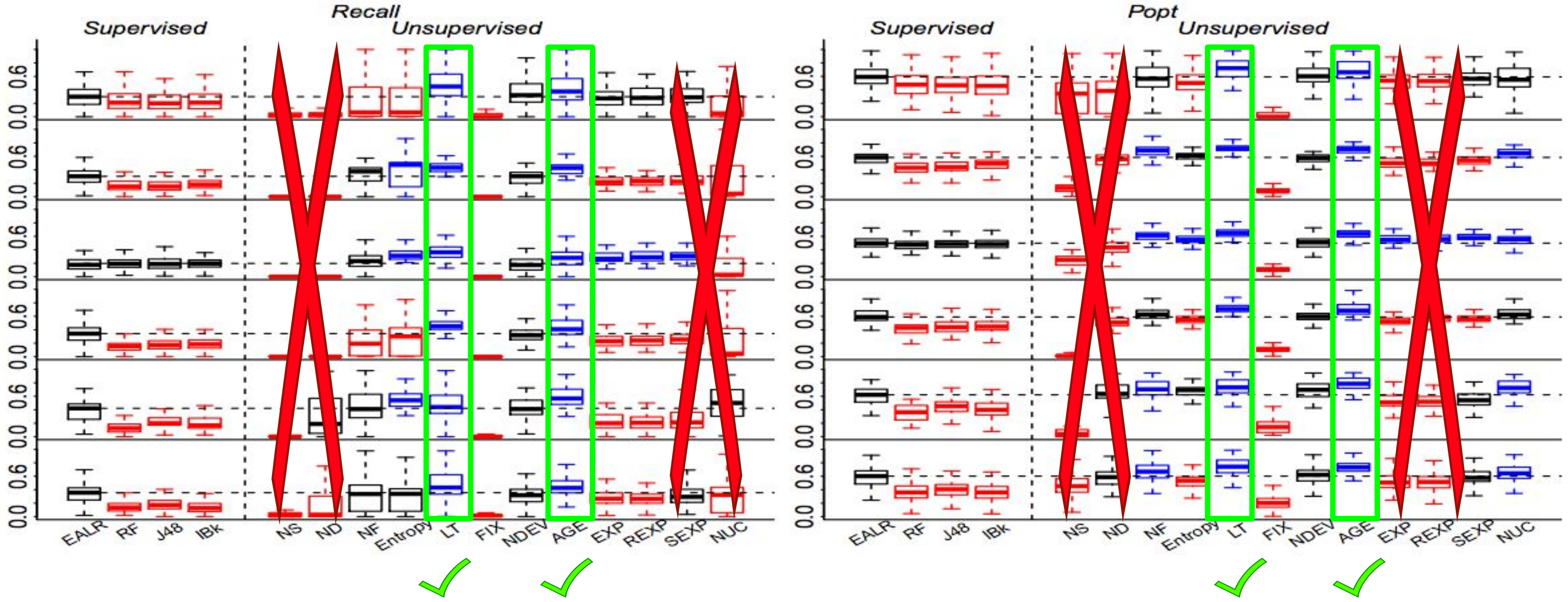
Research Questions

- All unsupervised predictors better than supervised?
- Is it beneficial to use supervised data?
- *OneWay* better than standard supervised predictors?

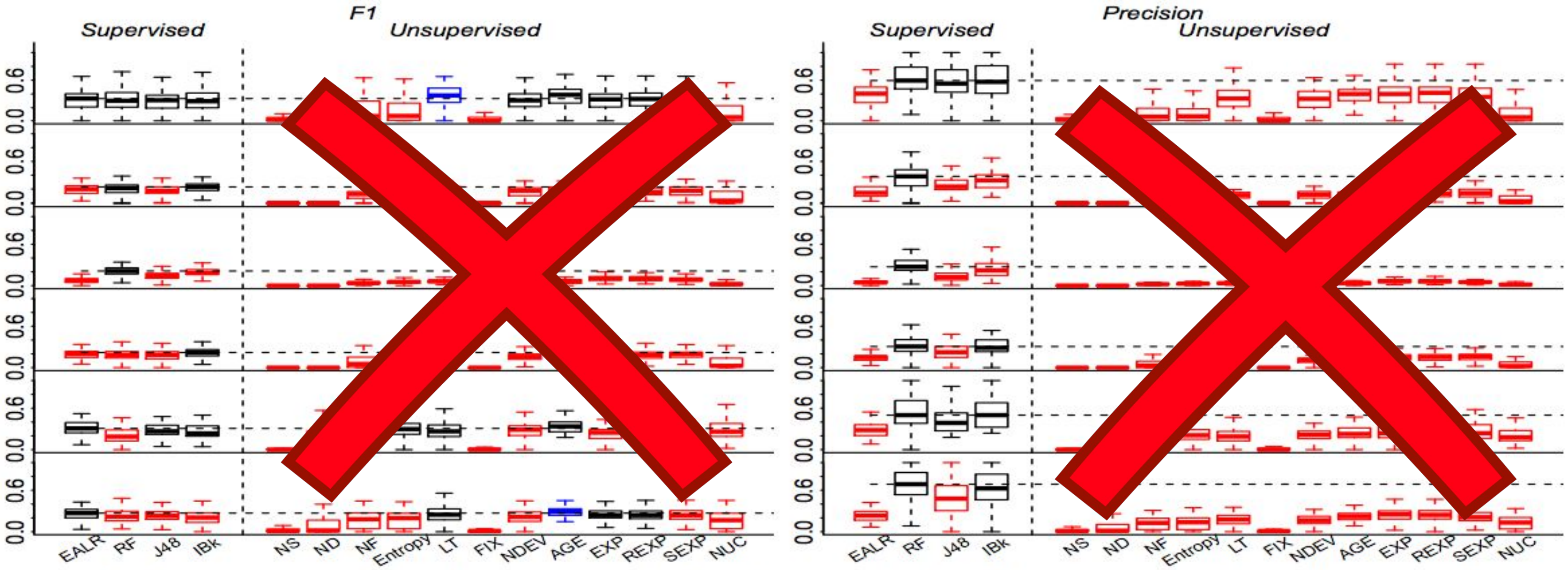
Research Questions

- All unsupervised predictors better than supervised?
- Is it beneficial to use supervised data?
- *OneWay* better than standard supervised predictors?

RQ1: All Unsupervised Predictors Better ?



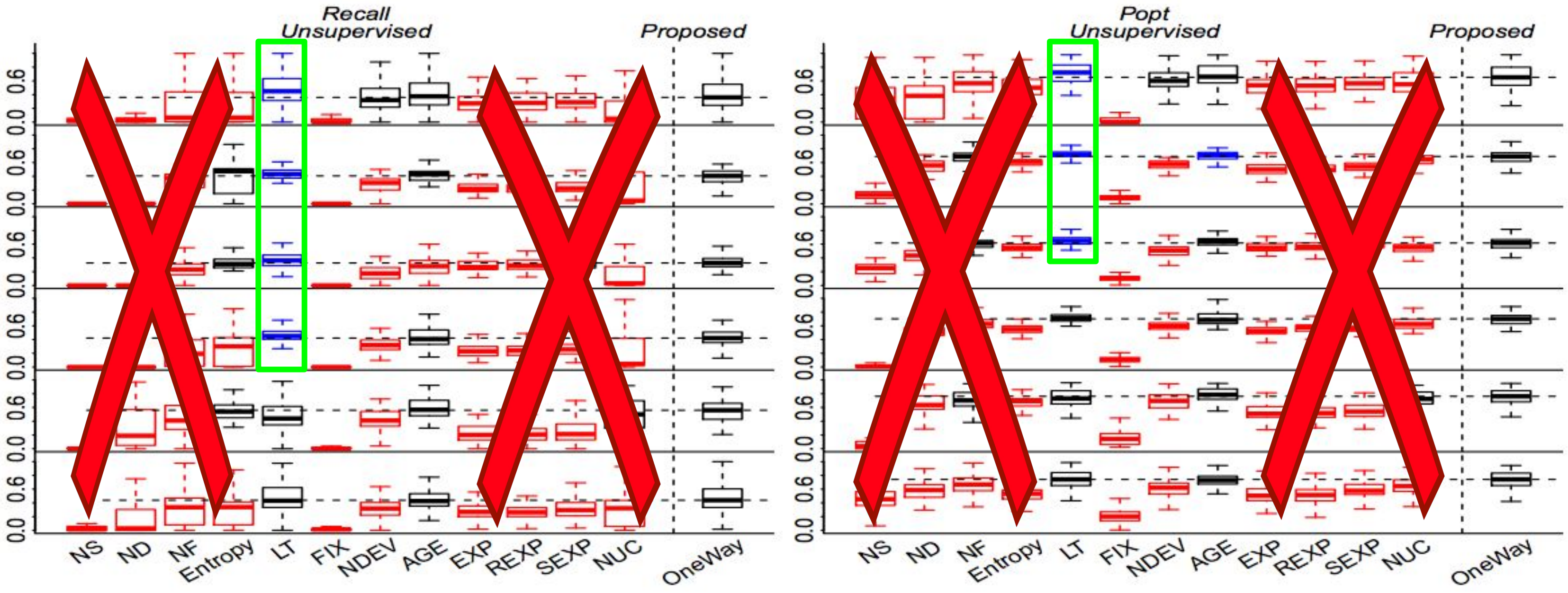
RQ1: All Unsupervised Predictors Better ?



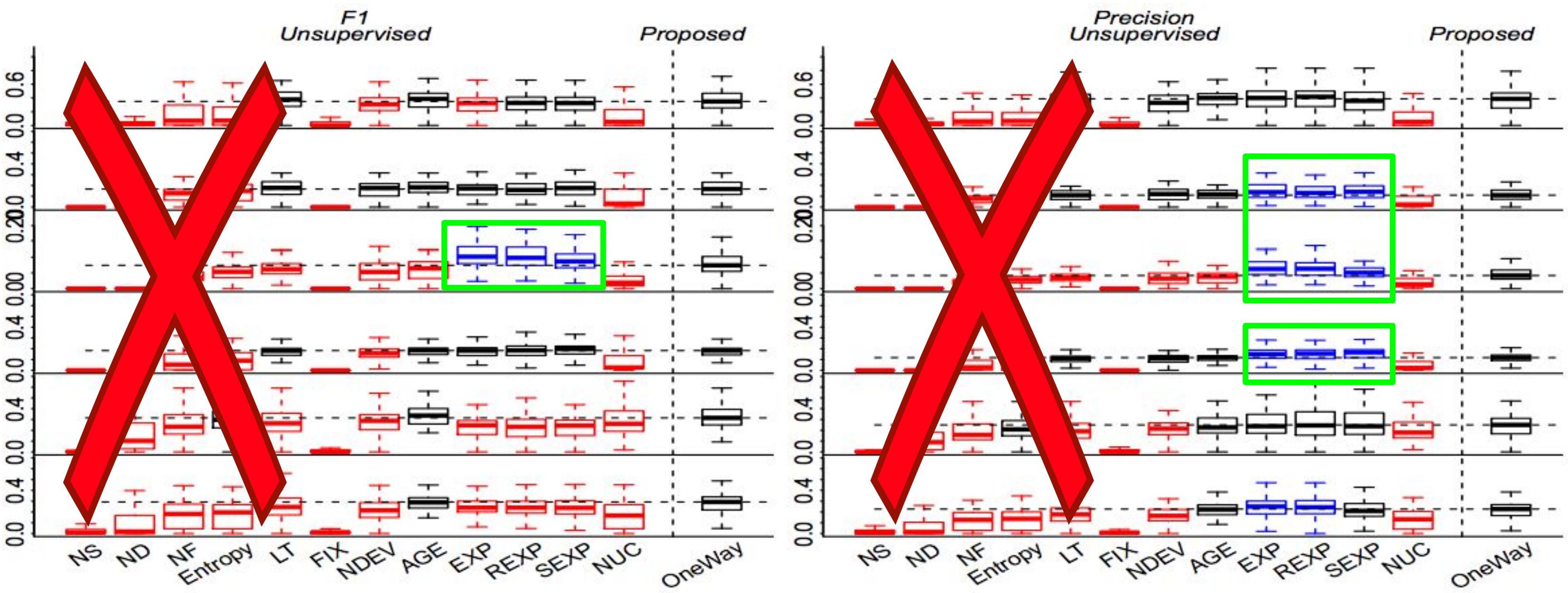
Research Questions

- All unsupervised predictors better than supervised?
- Is it beneficial to use supervised data?
- *OneWay* better than standard supervised predictors?

RQ2: Is It Beneficial to Use Supervised Data?



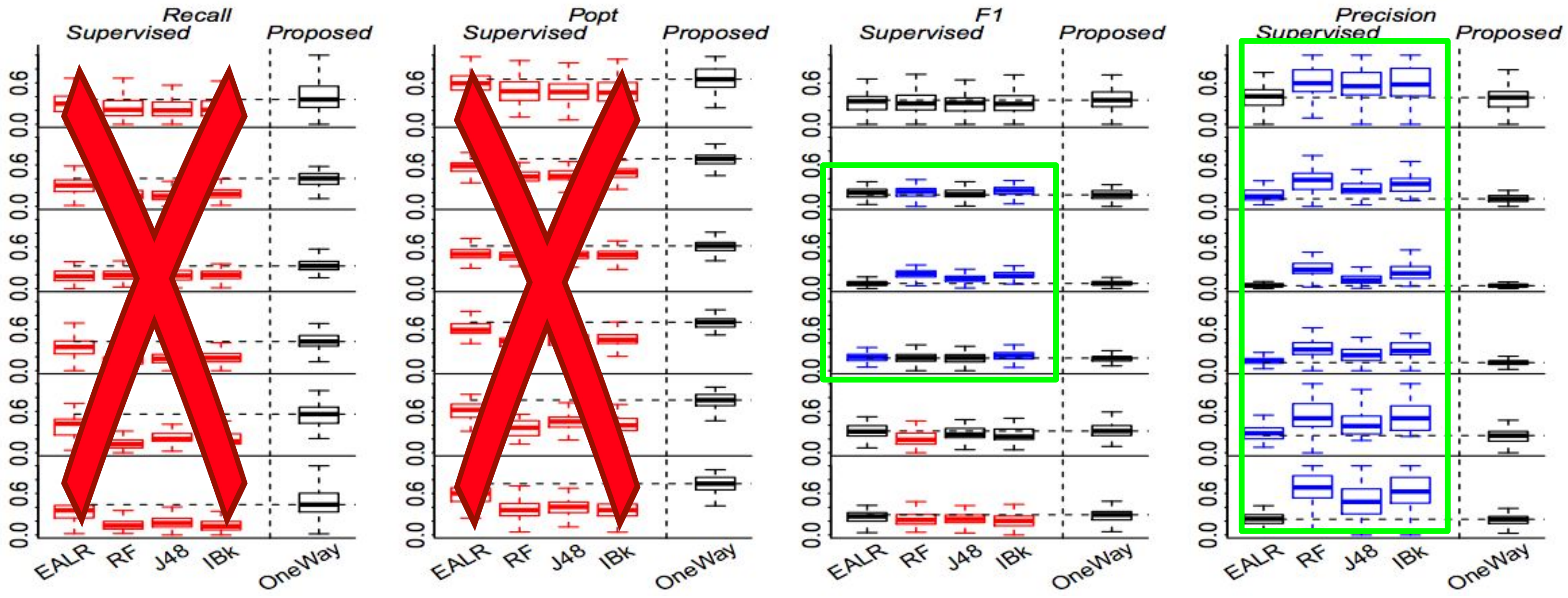
RQ2: Is It Beneficial to Use Supervised Data?



Research Questions

- All unsupervised predictors better than supervised?
- Is it beneficial to use supervised data?
- *OneWay* better than standard supervised predictors?

RQ3: *OneWay* Better than Standard Supervised Predictors?



Recall **Popt** **F1** **Precision**

My Thesis

Software analytics **should** be easier.

Software analytics **can** be easier. [Fu et al, IST 2016]

But it can be **very hard** to show it can be easier. [Fu et al, FSE 2017 A]

And, sometimes, it can be **too easy**. [Fu et al, FSE 2017 B]

When to be simpler?

Wei Fu, Tim Menzies, Di Chen, and Amritanshu Agrawal. "**Building Better Quality Predictors Using ‘ε-Dominance’**." *Submitted to FSE’ 2018.*

"Many Roads Lead to Rome"

Similar learners:

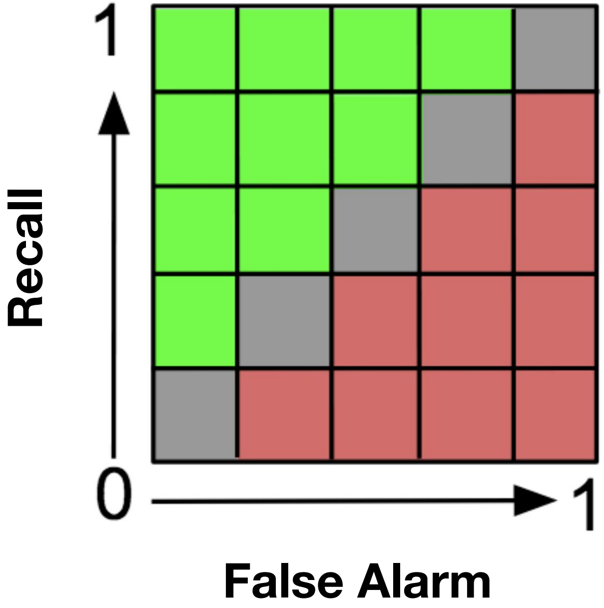
- Lessman et al[lessman'08]: 17/22 defect predictors are indistinguishable.
- Gohtra et al[Gohtra'15]: 32 defect predictors can be clustered into 4 groups.

If learners have a "result space"(recall vs false alarm):

- What "shape" of results spaces leads to "many roads"?
- Can we reverse engineer from that space a much simpler defect predictor?

Deb's principle of ε-Dominance

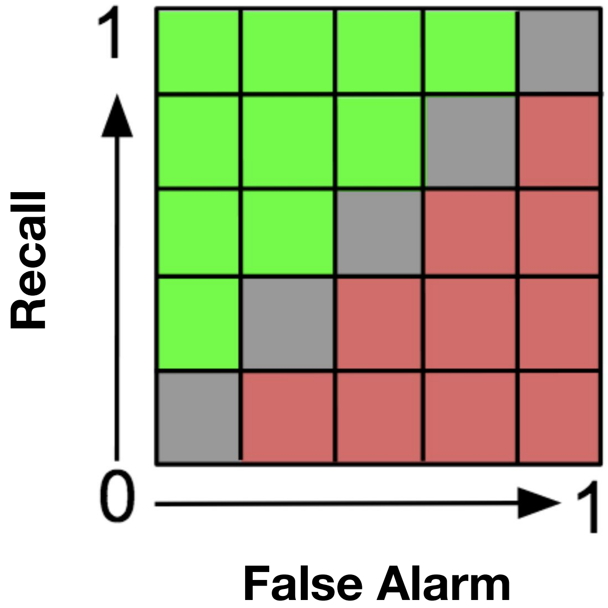
*If there exists some ϵ value below which it is useless or impossible to distinguish results, then **It is superfluous to explore anything less than ϵ***



$\epsilon = 0.2$

Deb, Kalyanmoy, Manikanth Mohan, and Shikhar Mishra. "Evaluating the ε-domination based multi-objective evolutionary algorithm for a quick computation of Pareto-optimal solutions." Evolutionary computation 13.4 (2005): 501-525.

DART: Fast-and-Frugal Tree(FFT)



1. if cob <= 4	then false
2. else if rfc > 32	then true
3. else if dam > 0	then true
4. else if amc < 32.25	then true
5. else false	

We used d=4, 2^d=16 trees to explore the results space.

RQ1: DARTS Better than Established Learners?

Goal	Data	DART	SL	NB	EM	SMO
dis2heaven: (less is better)	log4j	23	53	51	56	48
	jedit	31	40	41	34	47
	lucene	33	40	44	44	71
	poi	35	36	57	70	45
	ivy	35	50	40	71	43
	velocity	37	61	40	49	60
	synapse	38	51	39	34	62
	xalan	39	55	55	70	68
	camel	41	60	52	44	71
	xerces	42	68	60	50	69

Goal	Data	DART	SL	NB	EM	SMO
P_{opt}: (*more is better*)	ivy	28	17	9	28	23
	jedit	39	10	9	16	17
	synapse	43	26	24	22	22
	camel	53	15	17	16	50
	log4j	56	19	22	16	23
	velocity	64	64	64	24	60
	poi	73	51	19	33	64
	lucene	81	43	27	20	80
	xerces	90	4	9	15	48
	xalan	99	11	15	100	51

* Ghotra, Baljinder, Shane McIntosh, and Ahmed E. Hassan. "Revisiting the impact of classification techniques on the performance of defect prediction models." Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, 2015.

RQ2: DARTS Better than Goal-Savvy Learners?

Data	*dis2heaven* (*less is better*)		P_{opt} (*more is better*)	
	DART	Tuning RF	DART	Tuning RF
ivy	35	56	28	28
jedit	31	35	39	39
synapse	38	57	43	48
camel	41	70	53	54
log4j	23	51	56	20
velocity	37	53	64	64
poi	34.8	27	73	74
lucene	33	35	81	80
xerces	42	70	90	94
xalan	38.7	36	99	99

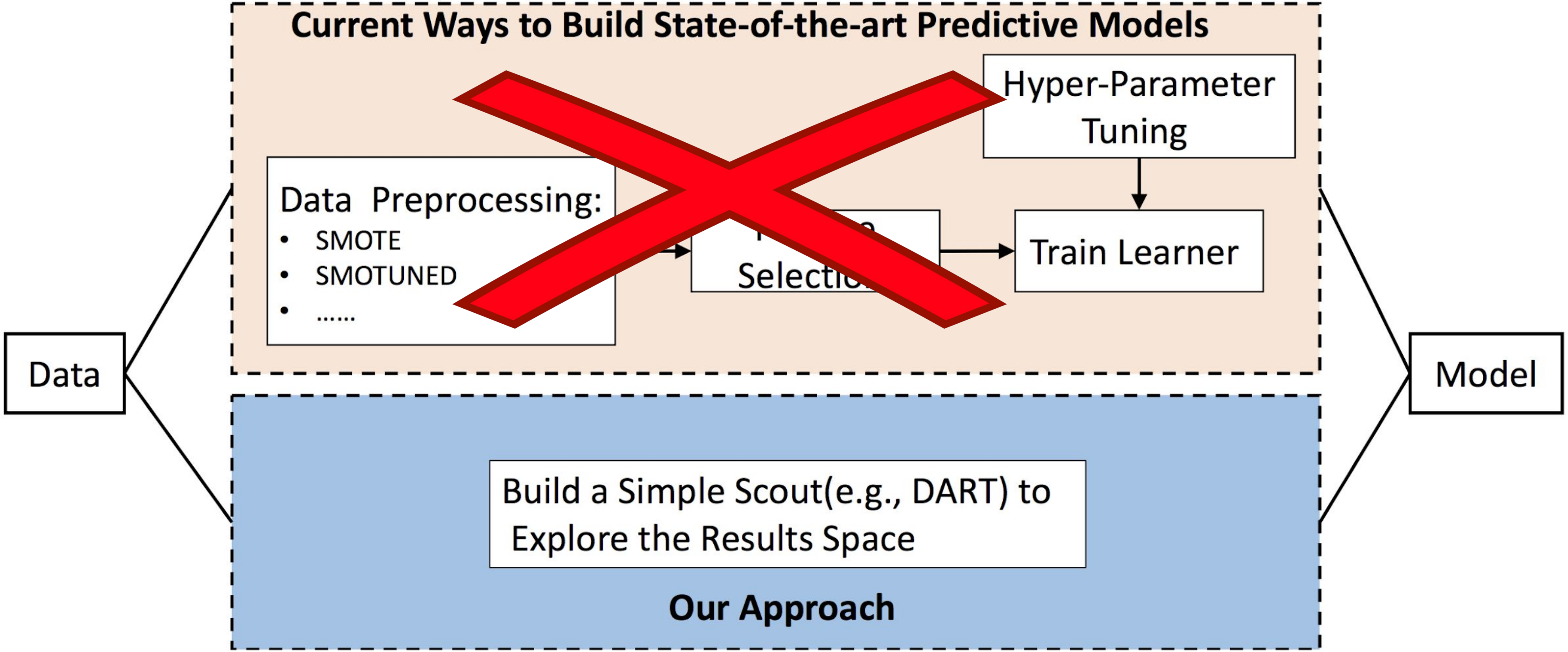
Fu, Wei, Tim Menzies, and Xipeng Shen. "Tuning for software analytics: Is it really necessary?." Information and Software Technology 76 (2016): 135-146.

RQ3: DARTS Better than Data-Savvy Learners?

Goal	Data	DART	KNN	SMO	NB	RF	SL	DT
dis2heaven: (less is better)	log4j	23	45	44	50	44	40	47
	jedit	31	45	52	41	39	44	40
	lucene	33	37	45	44	41	40	40
	poi	35	38	52	52	39	46	43
	ivy	35	37	46	36	39	37	40
	velocity	37	56	64	40	44	61	42
	synapse	38	36	47	36	42	37	42
	xalan	39	20	35	45	25	71	28
	camel	41	45	62	47	35	53	38
	xerces	42	45	67	52	52	53	53
P_{opt}: *(more is better)*	ivy	28	26	27	10	27	24	26
	jedit	39	3	17	6	10	4	24
	synapse	43	39	38	27	36	36	35
	camel	52.9	53	53	21	52	53	49
	log4j	56	27	50	24	33	44	44
	velocity	64	56	64	64	57	65	53
	poi	73	67	69	26	72	72	71
	lucene	81	45	49	27	49	42	53
	xerces	90	73	63	20	50	77	48
	xalan	99	99	98	24	93	100	88

*Agrawal, Amritanshu, and Tim Menzies. "" Better Data" is Better than" Better Data Miners"(Benefits of Tuning SMOTE for Defect Prediction)." arXiv preprint arXiv:1705.03697 (2017).

Conclusion


Current Ways to Build State-of-the-art Predictive Models
Hyper-Parameter Tuning
Data Preprocessing:
• SMOTE
• SMOTUNED
• ……
Train Learner
Data
Model
Build a Simple Scout(e.g., DART) to Explore the Results Space
Our Approach

Future of Future Work

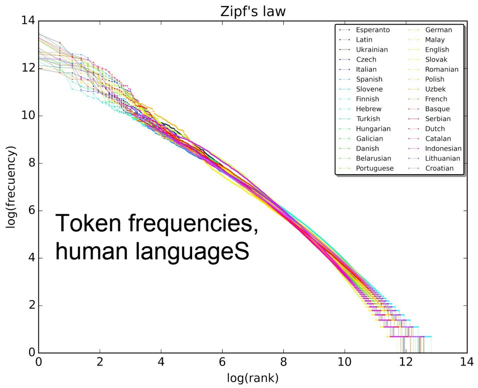
- Apply ε-Dominance to other software analytics tasks.
  - Text Mining
  - Issue closing time prediction
- Determine ε threshold
- Other criteria to simplify software analytics.

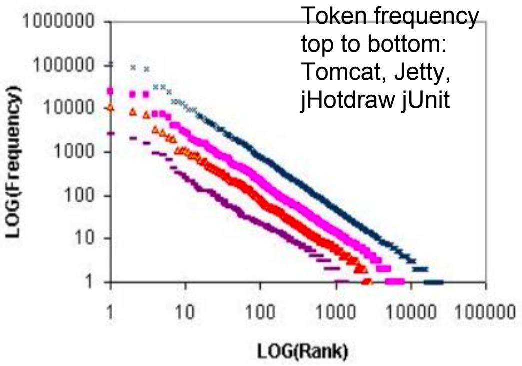
From Last Exam

Why study simplicity? *cost, speed*

When this won't work? *ε-Dominance*

What's the difference between SE/general data mining? *under-exploited simplicities*





Thank You!