Noname manuscript No.
(will be inserted by the editor)

Optimizing optimization via predictive models

Tim Menzies

Received: date / Accepted: date

Abstract In this special issue on predictive modeling, we study models for software analytics that predict the effectiveness of an optimizer. In turns out, that for many SE domain, it takes very little data to train a predictive model that can guess if an optimizer will like/dislike the next example. Using these models, software analytics can stop early (since this predictive model can be used for the subsequent reasoning).

To test this approach, we explore 27 challenge problems (ranging from from high-level decisions about software processes to low-level decisions abut how to configure video encoding software). When these problems have been previously explored by software engineering researchers, they required hundreds to thousands to millions of labelled examples. Here, we achieve significant improvements using just a handful of examples (sometimes, as few as 12 labels).

Other SE researchers have explored model building using limited labelling via unsupervised or semi-supervised learning, or active learning, or zero/few-shot learning. But to the best of our knowledge, this paper's success with only a handful of examples (and no large language model) is unprecedented.

We call this approach LITE since it is inherently different to HEAVY methods that (e.g.) need a large language model for some generative task (such as code translation). As shown in this paper, not all SE tasks are HEAVY. Hence it is prudent to research both LITE and HEAVY methods

This LITE approach is recommended for human-in-the-loop process where humans have to assess the examples (e.g. humans are generating the labels; or humans are assessing the examples used in building a model).

In accordance with the principles of open science, we offer the reproduction package of `XXX` to allow others to repeat/refute/improve these results.

Keywords Predictive Modeling · Search-based SE · Software Engineering

Tim Menzies
Computer Science, NC State
E-mail: timm@ieee.org

1 Introduction

This paper explores models for software analytics that predict the effectiveness of an optimizer. We will show that for many SE domain, it takes very little data to train a predictive model that can guess if an optimizer will like/dislike the next example. Using these models, software analytics can stop early (since this predictive model can be used for the subsequent reasoning). We call this approach "LITE":

> **LITE** = Low Input Tuning Enhancement: *improve a software system's behavior without having to spend much effort labelling local data.*

LITE addresses the *labelling problem*. One way to study some system "f" is to explore examples EGs $= (X, Y)$ where $X = f(Y)$. Here, X are the independent variables and Y are one or more numeric dependent goal variables. EGs are usually created without Y values and *labelling* is the process of reading X in order to compute Y. LITE assumes we can only label dozens, not thousands of EGs. As such, LITE is very amenable to human-in-the-loop reasoning where humans have to assess the EGs (e.g. humans are generating the labels; or humans are assessing the EGs used in building a model)

An alternative to LITE are HEAVY methods that assume "the more EGs the better"– an approach endorsed by many SE researchers; e.g. [10, 34, 42]. Some SE tasks are inherently HEAVY, such as the creating a large language model for use on generative tasks such as code translation [6]. That said, as shown in this paper, not all SE tasks need a HEAVY approach. Hence it is prudent to research both LITE and HEAVY methods.

To kick start the LITE research agenda, this paper, in §2, makes the business case for studying LITE-style reasoning. Next, in §3 we detail the labelling problem, and why we should avoid over-labelling. This is followed, in §5 by a discussion on why the problems addressed by LITE are not solved by current research into interactive SBSE, active learning, unsupervised and semi-supervised learning, and zero/few-shot learning. After that, §**??** describes `lite.py`, a tiny implementation of a multi-objective TPE (tree-structured Parzen estimator) that implements incremental model construction; i.e. build a model from a handful of labelled EGs (divided into "best" and "rest") then use than model to select what to label next.

§**??** describes case studies from the SE optimization literature. Prior work in SE needed thousands to millions of labels for these studies. But as we shall see in §**??**, `lite.py`'s significantly improves the default behavior seen in those problems, sometimes after just half a dozen labels. To the best of our knowledge, this is the fewest labels ever used for these tasks in SE.

Finally, to encourage further work, §**??** offer twenty challenge problems that could be readily explored using `lite.py` and the 27 case studies. Our hope is that numerous researchers repeat our experiments and discover methods that are superior to the one proposed here. Paradoxically, this means this paper will be a success if it is quickly superseded.

2 The Business Case for LITE

Enabling humans to explore the important potential behaviors of a software model is an open and important issue. For example, the more we use AI in SE, the more code will be auto-generated. The more we auto-generate code, software engineers will spend less time writing new code and time reviewing code, written by someone or something else (the internals of which they may not understand). The less we understand code, the more we will use components black-boxes where once a system is assembled, its control settings are tuned. In this scenario, it becomes very important to reduce the human effort and CPU effort required for that tuning.

We predict that enhancing our ability to reviewing models will become increasingly important in SE. In "Flaws of policies of requiring human oversight" [21], Ben Green notes that many recent policies require humans-in-the-loop to review or audit decisions from software models. E.g. the manual of the (in)famous COMPAS model notes the algorithm can make mistakes and advises that "staff should be encouraged to use their professional judgment and override the computed risk as appropriate" [39].

Cognitive theory [47] tells us that humans use heuristic "cues" that lead them to the most important parts of a model before moving on to their next task. But when humans review models, they can miss important details. Such cues are essential if humans are to tackle their busy workloads. That said, using cues can introduce errors: *...people (including experts) are susceptible to "automation bias" (involving) omission errors—failing to take action because the automated system did not provide an alert—and commission error* [21]. This means that oversight policies can lead to the reverse of their desired effect by *"legitimizing the use of faulty and controversial algorithms without addressing (their fundamental issues"* [21].

Sadly, there are too many examples of such errors:

- Widely-used face recognition software predicting gender & age, has a much higher error rate for dark-skinned women compared to light-skinned men [3].
- Amazon's delivery software became biased against black neighborhoods [1].
- Google Translate has gender bias. "She is an engineer, He is a nurse" translates Turkish then English as "He is an engineer, She is a nurse" [12].
- The COMPAS model guesses that black defendants as future criminals twice as much as whites [2]. It took years to find and fix this bug (and that difference is definitely a bug since it can be fixed [14] while maintaining the same levels of recall on actual recidivism).
- For other examples, see [20, 38, 44].

To forge an effective partnership, humans and artificial intelligence (AI) need to understand each other's strengths and limitations. Software can explore a very large space, on pre-determined criteria. But humans can offer important and useful insight, but only over a small number of instances. We are interested in tools like LITE since they do not make excessive demands of humans reviewing software systems.

3 Problems with Labelling

In SE, the usual case is that is very cheap to find X values but far more expensive to find Y. For example, by mining Github, it is possible to discover all the X distributions of code size, number of dependencies per function, etc. But it can be far more complicated to find the associated Y values like "how much this could be sold on the market?" or "what is the time required to build this kind of software?". For example, the SZZ algorithm reports what code is associated with a bug fixing commit. Any number of papers report the complexity of making that determination [22, 43].

Many labels are naive or faulty. Defect prediction researchers [13,23,26,28, 36] often label a commit as "bug-fixing" when the commit text uses words like "bug, fix, wrong, error, fail, problem, patch ." Vasilescu et al. [50, 51] warns that this can be somewhat ad-hoc, particularly if researchers just peek at a few results, then tinker with regular expressions to combine a few keywords. Also, when Yu et al. [56] explored labels from prior work exploring technical debt, they found that more than 90% of labels marked as "false positive" was true. In the SE literature, there are similar reports where data labels error have corrupted the majority of the EGs for security bug labeling [53]; for labeling false alarms in static code analysis [27]; or for software code quality [46].

Not only can labelling be error prone, it can also be very expensive. For example, Tu. et al. [48] studied 714 software projects, including 476K commit files. After an extensive analysis, they proposed a cost model for labeling their EGs. Assuming two people checking per commit, that data would need three years of effort to label their EGs.

Since labelling is error-prone, expensive and/or time-consuming, we explore methods that use as few labels as possible. One reason to use minimal labels is that this simplifies the review process described in the last section. Specifically, is is simpler to audit a system when that review means reflecting over the handful of EGs used used to build the model.

Before going on, we offer one piece of nomenclature about labels. For an EG with goal labels Y, each goal has a range *lo .. hi* and so can be normalized 0..1. Each goal has a *heaven* of 0,1 if that goal is to be minimize or maximized (respectively). We say an example has a distance to heaven (*d2h*) equal to the Euclidean distance of the goal values to a mythical best value. For example, if we want to minimize *bugs* and *features*, and these have the range 0 to 100, then for an example with 30 *bugs* and 80 *features*:

$$d2h((X,Y)) = dist(heaven, norm(Y)) = \sqrt{(0-.3)^2 + (1-.8)^2}/\sqrt{2} = 0.26 \qquad (1)$$

Note that *smaller* values of *d2h* are *better*. Also we dividing the distance by the $\sqrt{|goals|}$, to ensure *d2h* has a range zero to one.

4 How Many Labels are Enough?

The previous section warned that it is unwise (and expensive) to expect or rely on large libraries of labelled data. This section asks "at what rate can we expect to collect high-quality labels?". Here we explore some knowledge acquisition exercise where SMEs (human subject-matter experts) have to carefully justify all their decisions (perhaps even to a panel of other experts).

Acquisition of high-quality labels from humans experts can be remarkably slow. Valerdi [49] documents the effort associated with getting a panel of experts to agree on the labels of 60 software project effort estimation examples, where each example has 20 attributes. That study required 3*three hour sessions, spread out over one working week.

Other research offers similar numbers to Valerdi. Researchers in the field of *repertory grids* conduct structured interviews where humans justify attributes and attribute settings for random subsets of 3 examples, drawn from a larger set.

- Easterby-Smith [18] advises "keep the grid small. A grid containing ten elements and ten constructs may take two hours to complete. Larger grids may take substantially more time".
- Kington [29] agrees, saying that it takes humans an hour to reflect over 16 examples with 16 attributes.

Anyone who has worked with real-world SMEs know that experts are experts precisely because their expertise is valuable to the organization. This means that, by definition, the SMEs needed for high quality labelling may often called away to other tasks. In our experience:

- Initially, this kind of senior-level expertise can only be accessed for a few hours per month. In these initial sessions, SMEs might be interested in your techniques, but will soon abondon you if there is nothing to show for their valuable time.
- More access to more labelled data can be achieved– but only if substantive results are achieved with the few labels collected from the initial sessions.

Accordingly, the goal of this paper is to understand what happens if we can only access 80,40,20 or even as few as 10 labels.

5 Methods for Labelling Less

Other SE researchers have explored model building using a few labelled EGs. But as we shall see in this section, (with some caveats) those method are rarely applied on as little as 80,40,20 or 10 labels.

There are two kinds of algorithms that make decisions about software without new labels from the test domain. **Zero-shot learners** use the background knowledge of a large language model to make decisions without needing new labels [7]. Zero-shot learners works in domains where there exists an appropriate language language model (which is *not* the situation for our case studies).

Apart from zero-shot learners, traditional **unsupervised learners** use a domain heuristic to classify examples by peeking at the independent variables: E.g. Nam et al. successfully predict for defective modules by looking for classes that are unusually large [37]. For our case studies, we know of no such domain heuristics.

Another technique for labelling less is to recognize some system condition under which an EG can be unequivocally labelled as "fail"; e.g. (a) if a test generates a core dump or (b) some metamorphic predicate [16] argues that, say, "small changes to inputs should not cause large changes to output":

- The value of this approach is that such a domain-general labelling oracle might be relevant to other applications.
- On the flip side, the more general the test oracle, the less it connects to the specific goals of the local users. For example, in the case studies shown below, we test for user-supplied local goals such as "if we implement this requirement next, then some other team will not stand idle waiting for some other function we were meant to implement".

Another class of algorithms use less labels by, say, labelling just a few EGs, then propagate those values to other near-by EGs. Such **semi-supervised** learners have successfully reasoned over 10,000s records, after labelling just 1 to 2.5% percent of the EGs [8,33]. While a useful approach, these method still needs 100s to 1000s of labels.

Human-in-the-loop **active learners** minimize human labelling effort by updating their models each time a human offers an opinion on an EG. Still, even with these tools, the labelling effort can be excessive. Yu et al.'s active learners need 20% of the data labelled, which for (e.g.) 28,750 files in Firefox means labelling 5,750 EGs [57]. Similar values of 10 to 20% labelling have been reported by other active learning researchers in SE: see Wu et.al [52].

Researchers in **interactive-search-based SE** explore human-in-the-loop optimization where humans serve as the oracle that judges if one EG is better than another. The problem with iSBSE is that when humans are asked too many questions, they start making mistakes due to *cognitive fatigue* [45]. One solution to this problem is to learn a *surrogate model* from a small sample of EGs, then querying the surrogate for labels [11]. But even with technologies like surrogates, our reading of the iSBSE research [9,11,32] is that community still assumes they can label hundreds of examples, or more.

In the SE literature, one area close to this paper is **few-shot learning**. This is a technique that uses a few dozen labeled EGs (sometimes as few as ten) to convert a general large language models into some specific tool; e.g. parsing test case output [31] or translating functions into English [6]. But like zero-shot learning, few-shot learning only works where, in the domain being explored, there exists a functioning large-language model. As we shall see, there are many domains in SE where this is not true.

5.1 Prior Work with "SWAY"

In our own prior work, the closest approach we have to LITE are the recursive random projections of SWAY [15]. SWAY was inspired by algorithms that clustered data using PCA (principle components analysis [40]). The advantage of PCA is that some data sets are better described by synthesized attributes that combine the influences of raw attributes. But standard PCA can only handle numeric data so SWAY combined the Aha et al. [5] distance metric[1] with the FASTMAP heuristic[2] [19] to recursively bi-cluster EGs using the independent attributes. Formally, SWAY belongs to the family of Nyström algorithms that approximate PCA, by various methods [41].

SWAY implement optimization as a binary chop through a space of EGs clustered as per the last paragraph. At each level of the recursion, SWAY uses FASTMAP and Aha to find two distant examples A, B. It then sorts them into "best" and "rest" using Equation 1. SWAY then recurses on the examples nearest the "best". On recursion, when looking for two more distant points, SWAY reuses (without relabelling) the "best" EG from the parent. This means that SWAY needs two evaluations for the root split (since, there, there is no parent cluster), then one new labelling for each level of recursion.

SWAY terminates when leaf classes contain less than some user-specified point; e.g. $\sqrt{N}$. On termination, it returns the last ever "best" example found by the algorithm. For clustering $N = 10,000$ examples into bins of size $\sqrt{N} = 100$, SWAY requires nine labels.

SWAY is a "greedy search"; i.e. it decisions using purely local information. Such a search is susceptible to premature commitments since it might prune good EGs without enough reflection.

An alternative to SWAY's greedy search is the predictive modeling approach of `lite.py`. That algorithm rebuilds its knowledge every time is sees a new EG, then searches afresh over the unlabelled data from the next thing to label., The experiments of this paper will show that the predictive modeling of `lite.py` out-performs SWAY's greedy search.

6 Predictive Modeling with `lite.py`

XXX need an example here

[1] Aha et al. [5] suggest that points $(X_1, Y_1), (X_2, Y_2)$ are separated by $dist(X_1, X_2) = \left(\sum_i \Delta(X_1', X_2')^2\right)^{0.5}$. For discrete values, $\Delta(a, b) = 0$ *if* $a == b$ *else* 1. For numeric values, $\Delta(a, b) = abs(\hat{a} - \hat{b})$ where $\hat{a}$ is a normalized to the range 0..1 for min..max. For missing values, we use values for a, b that maximizes the computed distance.

[2] To recursively bi-cluster data, first pick any EG at random; then use the Aha metric to find a remote example, called A. Next, find the example B furthest from A and label A, B "best" and "rest" using Equation 1. If every other EG have a distance a, b to A, B, then all those points fall on a line between A, B at $x = (a^2 + c^2 - b^2)/(2c)$, where $c = dist(A, B)$. After splitting on median x, then this algorithm recursive on "best" half. This approach is fast since each level of this recursion only requires $2N$ distance calculations.

As stated in the introduction, one of the stated aims of this paper is to encourage much experimentation with a LITE approach to software analytics. Hence it is apropos to note that the source code for `lite.py` (and, indeed, SWAY) is very short.

Oune problem with If DODGE is out-perform by "compare with 30 random EGs", then why go any further? Why not recommend that all optimizations of local data just tries 30 random EGs, then stop?

For two reasons, we need to do better than "use 30 random". Firstly, random search returns nothing at all except a set of EGs, sorted by some criteria. This is undesirable. Ideally, we want an optimization study to return a generalizable model that can be applied to future EGs:

- Without such a generalizable model, then if ever we have to repeat an optimization study, that second study will take as much effort as the first study.
- But with such a generalizable model, if that second study reuses the model learned from the first study, the it may be possible to avoid any further labelling.

Secondly, another reason to do better than "use 30 random" is that 30 labels may be too many. This is particularly true if those labels are collected from SMEs (human subject matter experts).

The more society depends on software, the more likely it becomes that some organizational policy or some legislative requirement will demand that that software gets evaluated. Green [21] comments that when faced with large and complex problems, cognitive theory [47] tells us that humans use heuristic "cues" to lead them to the most important parts of a model. Such cues are essential if humans are to reason about large problems.

That said, using cues can introduce their own errors: *...people (including experts) are susceptible to "automation bias" (involving) omission errors - failing to take action because the automated system did not provide an alert - and commission error* [21]. This means that oversight can lead to the opposite desired effect by *"legitimizing the use of faulty and controversial (models) without addressing (their fundamental issues")* [21].

By "faulty and controversial models", Green refers to the long list of examples where detrimental models were learned via algorithmic means. For example, Cruz et al. [17] lists examples where:

- Proposals from low-income groups are five times more likely to be incorrectly ignored by donation groups;
- Women can be five times more likely to be incorrectly classified as low-income;
- African Americans are five times more likely to languish in prison until trial, rather given the bail they deserve.

These are just a few of the many reported examples [44][3] of algorithmic discrimination. For another example, the last chapter of Noble [38] describes how

[3] See also `http://tiny.cc/bad23a`, `http://tiny.cc/bad23b`, `http://tiny.cc/bad23c`

a successful hair salon went bankrupt due to internal choices within the YELP recommendation algorithm.

Mathews [25] argues that it is not surprising that so many She notes that everyone seeks ways to exploit some advantage for themselves. Hence, we should expect that the software we build to discriminate against some social groupings;

> *"People often think of their own hard work or a good decision they made. However, it is often more accurate to look at advantages like the ability to borrow money from family and friends when you are in trouble, deep network connections so that you hear about opportunities or have a human look at your application, the ability to move on from a mistake that might send someone else to jail, help at home to care for children, etc. The narrative that success comes from hard work misses that many people work hard and never succeed. Success often comes from exploiting a playing field that is far from level and when push comes to shove, we often want those advantages for our children, our family, our friends, our community, our organizations."*

Hence I assert that unfairness is a widespread issue that needs to be addressed and managed. Specifically, we need to ensure that a software system created by one group, A, can be critiqued and modified by another group, B. We say that such a review has several properties:

It will usually be conducted under some from of time-pressure. The kind of

The more and more time reviewing code (from other sources). The current evidence is that such reviews often make mistakes, perhaps since they are often done under some time constraints.

Hence this paper proposes a new benachmark problem: how quickly can we learn ways to change a rpogram's behavior, given li perhaps due Tomorrow's software engineers will spend much time reviewing model written by someone or something else (e.g. an LLM). Given divergence in requirements, this review process will need to balance multiple, possibly competing, goals. To be time-effective, these reviews should also complete without having to label many examples.

This task can characterized as a multi-objective semi-supervised optimization task. This paper explores dozens of such tasks The LAB tool (the Lightweight Adaptive Balancer) was able to use tree-structured Parzen estimators to find large changes that improved default behavior after just seeing 24 labels. This result held even when reviewing tens of thousands of examples.

We hence conclude that tree-structured Parzen estimation tools like LAB offer much support for the model review task. It is especially recommended when there is very little locally labelled data.

LITE and SWAY both use the same underlying object model described in Listing 3. That code uses the control parameters defined in a class called `the` ❶ and referenced in the code using the idiom (e.g.) `the.file`. The semantics of those values are discussed later in this paper.

```
class the:
   # misc settings
   file="data.csv"  # data file
   p=2              # in distance calcs, use a Euclidean distance measure

   # bayes classifier settings
   m=2              #  kludge for hanlding low frequency attribute values
   k=1              #  kludge for handling low frequency classes

   # smo settings
   best=0.5         # the n**best labelled items are "best"
   upper=0.8        # after sorting unlabbeled examples, keep top 80%
   label=4          # initially label 4 examples
   Label=16         # label at most another 16 examples

   # sway settings
   Far=0.95         # avoid outliers when seeking distant egs; don't go all the way
   some=256         # only consider some examples when seeking distant examples
   Min = 0.5        # recursion stops at N^min

   # sway2 settings
   dive=50          # at first, stop at leaves of size 50
   Deeper=4         # next tim around, stop at leaves of size 4
```

Listing 1: **Global settings:** The code uses control parameters that referenced in the code using the idiom (e.g.) "`the.file`". The semantics of those values are discussed later in this paper.

```
class SYM:  # Place to summarize a stream of symbols
  def __init__(self,name=" ",column=0): ❶
    self.name, self.column, self.n, self.has = txt, column, 0, Counter()

  def add(self,x)   : ...   # Add one more count for  symbol `x` ❽
  def dist(self,x,y): ...   # Using Aha, return  distance between two syms ❸
  def like(self,x,m,prior): # Return likelihood of x is in self  ❹
    return (self.has.get(x, 0) + m*prior) / (self.n + m) ❼

class NUM: # Place to summarize a stream of numbers
  def __init__(self, name=" ",column=0): ❶
   self.name, self.column, self.n = name, at, 0
   self.mu, self.m2, self.sd, self.lo, self.hi  = 0,0,0, 1E30, -1E30
   self.heaven =  0 if txt[-1] == "-" else 1 # if ending with "-", then minimized ❷

  def norm(self,x)   : ... # Return x normalized 0..1, self.lo..self.hi
  def dist(self,x,y) : ... # Using Aha, return distance between two numerics. ❸
  def add(self,x)    : ... # Use x to incrementally update lo,hi, mu, and sd ❽
  def like(self,x,*_):     # Return prob of x in a normal pdf of self.mu and self.sd ❼
    return 1/((σ+ϵ)*sqrt(2*π)) * exp(-(x-μ)^2/(2*σ^2+ϵ)) ❻
```

❶ NUMs and SYMs know column name and position, plus some other details.
❷ NUMs also know their "heaven" point; if minimizing, it is 0. Otherwise, it is 1.
❸ "`dist`" reports the delta between values using the Aha et al. formula from §5.1.
❹ "`like`" reports likelihood that "x" belongs to a distribution.
❺ As per Webb et al. [55]'s advice, the m offsets handle low frequencies.
❻ $\epsilon = 10^{-30}$ added to avoid divide-by-zero issues.
❼ As per Webb et al. [55]'s advice, the m offsets handle low frequencies.
❽ "`add`" update distribution. Which for numerics, is via Knuth's method; i.e.
$n \mathrel{+}= 1; d = x - \mu; \mu \mathrel{+}= d/n; m2 \mathrel{+}= d * (x - \mu); \sigma = (m2/(n-1))^{0.5}$

Listing 2: **Column classes**: summarizes columns into NUMs and SYMs. Some details omitted.

The code assumes EGs present themselves as a list of lists. The top list holds the column names, which select the column's role Upper case names are numeric columns and everything else is symbolic. ❷. Names ending in `+-` are goals to be maximized or minimized. ❸. The ideal heaven point of a goal being minimized is 0; else it is 1. ❹.

NUM and SYM are classes that summarize columns symbols and numbers respectively. These classes know their name and column position ❺.

Lists of NUMs and SYMs are updated by the DATA class whenever that class processes a new row ❻

Their summaries are updated by a add method. Thse

– x.add(y)

code runs over a DATA class that stores rows,

```
def isNum(s):  return s[0].isupper()  # numerics are upper case  ❷
def isGoal(s): return s[-1] in "+-"   # goal are to be maximimzed (+) or minimize(-) ❷
def isUsed(s): return s[-1] != "X"    # columns ending in "X" are to be ignored. ❷

class DATA: # Place to hold rows, summarized in the column headers
  def __init__(self, headerAndRows=[], order=False):
    self.names,*rows = list(headerAndRows)  ❶
    self.rows, self.x, self.y = [],[],[]
    self.cols = [(NUM if isNum(s) else SYM)(s,i) for i,s in enumerate(self.names)]  ❸
    [self.add(row) for row in rows]
    if order: self.ordered() ❺
    self.also(order)

  def also(self, order=False): ❹
    [(self.y if isGoal(c.txt) else self.x).append(col) for c in self.cols if isUsed(c.txt)]

  def add(self,row): # "Add row, summarized in column headers.  ❻
    self.rows += [row]
    [col.add(x) for x,col in zip(row,self.cols) if x != "?"] # Avoid 'dont know' values

  def clone(self, rows=[], order=False): # Return a DATA of same structure with new rows
    return DATA([self.names] + rows, order=order)

  def ordered(self): ...      # Sort rows by distance to heaven
  def d2h(self,row): ...      # Return Y-distance from row to best Y via Equation 1
  def dist(self, row1, row2): # Return X-distance row1 to row2
    d = sum(c.dist(row1[c.column],row2[c.column])**the.p for c in self.x) ❻
    return (d/len(self.x))**(1/the.p)

  def like(self,row,nall,nh,m=1,k=2): # Return log likelihood of row is in self  ❽
    prior = (len(self.rows) + k) / (nall + k*nh)
    tmp   = [col.like(row[col.colulmn], m, prior)  for col in self.x] ❼
    return sum(math.log(x) for x in tmp + [prior]) ❾
```

❶ Rows arrive as list of lists. Row1 defines column names. Other rows are the examples.
❷ Column names define the roles of each column.
❸ Columns are stored in `self.cols`.
❹ Further, independent and dependent columns are also store in `self.x,self.y`
❺ Optionally, rows can be sorted by the distance to heaven (Equation 1).
❻ **dist** is implemented using "dist" from Listing 2.
❼ "like" is implemented using "like" from Listing 2.
❽ As per Webb et al. [55]'s advice, the m, k offsets handle low frequencies.
❾ "like" returns log(like) for numerical methods reasons (we can use small negative numbers rather than very tiny floats).

Listing 3: **DATA class** stores rows (and their summaries in column headers). Some details omitted.

```
#class DATA continued
  def smo(self, score=lambda B,R: 2*B-R, fun=None):
    "Predictive modeling to predict if a new example is 'best' or 'rest'"
    def like(row,data):
      "Return log likelihood of row belonging to data"
      return data.like(row, len(data.rows), 2, the.m, the.k)
    def acquire(best, rest, rows):
      "Sort rows best to rest, keep the first (say) 80%"
      chop = int(len(rows) * the.upper)
      return sorted(rows, key=lambda r: -score(like(r,best),like(r,rest)))[:chop]
    #------------------------
    random.shuffle(self.rows)
    done, todo = self.rows[:the.label], self.rows[the.label:]
    data1 = self.clone(done, order=True)
    for i in range(the.Label):
      if len(todo) < 3: break
      n = int(len(done)**the.best + .5)
      top,*todo = acquire(self.clone(data1.rows[:n]),
                          self.clone(data1.rows[n:]),
                          todo)
      done.append(top)
      data1 = self.clone(done, order=True)
    return data1.rows[0],len(data1.rows)
```

Listing 4: Example from the Lua manual

```
#class DATA continued
  def near(self, row1, rows=None): # Return rows, sorted by dist to row1
    return sorted(rows or self.rows, key=lambda row2: self.dist(row1,row2))

  def far(self, rows, sortp=False, last=None): # Get 2 distant items, maybe reusing last
    n     = int(len(rows) * the.Far)
    left  = last or self.near(random.choice(rows),rows)[n]
    right = self.near(left,rows)[n]
    if sortp and self.d2h(right) < self.d2h(left): left,right = right,left
    return left, right

  def half(self, rows, sortp=False, last=None): # divide data by distance to 2 distant egs
    def dist(r1,r2): return self.dist(r1, r2)
    def proj(row)  : return (dist(row,left)**2 + C**2 - dist(row,right)**2)/(2*C)
    left,right = self.far(random.choices(rows, k=min(the.Half, len(rows))),
                          sortp=sortp, last=last)
    lefts,rights,C = [],[], dist(left,right)
    for n,row in enumerate(sorted(rows, key=proj)):
      (lefts if n < len(rows)/2 else rights).append(row)
    return lefts, rights, left

  def sway(self, rows=None,stop=None,rest=None,evals=1, last=None): # Recurse to best half
    rows = rows or self.rows
    stop = stop or 2*len(rows)**the.Min
    rest = rest or []
    if len(rows) > stop:
      lefts,rights,left  = self.half(rows, True, last)
      return self.sway(lefts, stop, rest+rights, evals+1, left)
    else:
      return rows, rest, evals, last

  def sway3(self): # SWAY twice. Second time, use rows found the first time
    random.shuffle(self.rows);
    rows1, rest1, evals1, __   = self.sway(self.rows, the.dive)
    rows2, rest2, evals2, last = self.sway(rows1, the.Deeper)
    return  rows2, rest1+rest2, evals1 + evals2 + 3, last
```

Listing 5: Example from the Lua manual

Supervised learning needs labelled data. This paper looks for alternatives to supervised learning since, as listed here, the SE literature often reports that labelling data is expensive, time-consuming and error-prone.

- Generating labels is expensive. For example, Tu et al. [48] cost models report that it would take three years relabel the data from the 714 software projects studied in their last publication.
- Not only is labelling expensive, it also is chronically error prone. Numerous labelling errors have been reported in widely used data set for technical debt [56] , security bugs [53]; for static code analysis [27]; or for software code quality [46].
- Methods that use no labels (*unsupervised learners*) have reported success in only limited domains such as defect prediction [54];
- Methods that label just a few examples, then propagate those values to other examples) have successfully reasoned over 10,000s records, after labelling just 1 to 2.5% percent of the data [8, 33]. But in those domains, these *semi-supervised learners* still needed labels for 100s of records.
- Human-in-the-loop *active learners* minimize human labelling effort by updating their models each time a human offers an opinion on an example. Still, even with these tools, the labelling effort can be excessive. Yu et al.'s active learners need 20% of the data labelled, which for (e.g.) 28,750 files in Firefox means labelling 5,750 files [57]. Similar values of 10 to 20% labelling have been reported by other active learning researchers in SE: see Wu et.al [52].
- Zhang et al. Devanbu et al. report that a few dozen full labeled examples (sometimes as few as ten) are enough for *few-shot learning* to convert large language models into effective tools for parsing test case output [31] or translating functions into English [6]. But few-shot learning only works where, in the domain being explored, there exists a functioning large-language model. As we shall see, there are many domains in SE where this is not true.

Another way to look at the labelling problem is not "how much data do we need" but "how fast can we generate high-quality labels?". To explore that question, we first must clarify the phrase "high-quality label".

As an example of a "low-quality labelling scheme", consider two graduate students separately skimming over over a stream of records, adding or checking labels. At the end of the session, the labelling teams meets to discuss and resolve any disagreements in their labelling. In this way, 100s to 1000s of records can be labelled in a week. Having funded many such sessions[4], I conjecture that one of the reasons of the above labelling errors is that this approach is so tedious that the labelling team often makes errors.

As an example of "high-quality labelling scheme", consider labelling exercises run by Richardo Valerdi as he collects data from a panel of subject matter about software cost estimation. One such panel is described in

Various semi-supervsed The point of this apepr is that ll of the above are too much. human time is limited. labelling is epextensive and error prone. predict a change in the maphsis of analutics from model construction to modl reivew.

[4] Using the power of pizza.

- very time constrained

- very data constrained

- releatively infrequent (so review session 1 had better produce a moedl taht can be used as a surrogate when the reviewers are unavailable) .

Need a new kind of anaytics, once that is based on very little labels. Based on experience with Promise, can achieve.

Given advances in large language models, tomorrow's software engineers will spend much time reviewing software written by someone or something else (e.g. an LLM). This review may be part of some legislated process where a review board of stakeholders (who are unfamiliar with the internals of the software) is required to decide how best to use some software system. Given divergence in stakeholder requirements, this review will need to strike a balance between multiple, possibly competing, goals.

XXXXFurther if tomorrow is anything like today, the review will be have to be completed quickly before reviewers are pulled aside for other tasks.

We hece propos that one fugure trend in software analtics, which is poolry supprot b current methods is model review.

To say that another way, we say that one way to formalize the model review task as a

> *Multi-objective semi-supervised optimization task that can access only a a handful of examples (say, dozens, not hundreds).*

This paper asks "is that task possible?". We offer baseline result based on a very simple toolkit (less than 400 lines of Pythn). XXXX. We shown that it, yes, indeed it is. In dozens of case studies (ranging from from high-level decisions about software processes to low-level decisions abut how to configure video encoding software) our LAB tool (the Lightweight Adaptive Balancer) was able to find large changes that improved default behavior of thee systems, after just seeing 24 labels. This result held even when reviewing tens of thousands of examples.

There are three intrestng aspects to this results:

- simplicity of the code

- stark contrast to standard practice

- far smaller than prior results

Lest the reader find that result improbable, we offer some a literature review and some maths showing that (a) there is some precedent in the SE literature for this result, albeit in other domains; (b) there are mathematical reasons to believe that with just few dozen labels, it is possible to be 95% confident of finding solutions that are statistically indistinguishable from the best solution.

We hence conclude that tree-structured Parzen estimation tools like LAB offer much support for the model review task. It is especially recommended when there is very little locally labelled data.

7 Background

7.1 the Problem of Model Review

The core motivation from this work is to find algorithms that support model review, with very little information.

mcintosh2017fix,rahman2013sample,amasaki2020cross

7.2 Big Data

A commonly expressed belief in machine learning and software analytics is that the more data, the better. For example:

- "Long-term JIT models should be trained using a cache of plenty of changes" [34];
- "..as long as it is large; (prediction performance) is likely to be boosted more by the sample size" [42];
- "It is natural to think that accumulating multiple releases can be beneficial because it represents the variability of a project" [10].

Yet sometimes, this "data-heavy" approach is not possible. As Devanbu comments:

7.3 Few-Short Learning

7.4 Active Learning

Over the years, there have been many results where very simple models performed exceptionally well in both the general AI literature [24,30] as well as in software analytics [4,35]. Using those results, it is possible to propose a "small data" style of analytics that relies on very simple models.

7.5 Why Study Simplifies

Why is it important to study simple models? We answer that question in three parts. Firstly we say that the models generated by sofytwre analytics need to be reviwed. Secondly, using that cognitive psychologcal theory argues that smaller models are more comprehensible. Based on those results, we propose a new style of software analytics where the goal is to achieve the most, with less resources.

This paper asks: can do something analogous to few-shot learning without an a large language model (LLM)?

We explore this since, given advances in large language models, tomorrow's software engineers will spend much time reviewing software written by someone or something else (e.g. an LLM). This review may be part of some legislated process where a review board of stakeholders (who are unfamiliar

with the internals of the software) is required to decide how best to use some software system. Given divergence in stakeholder requirements, this review will need to strike a balance between multiple, possibly competing, goals. Further if tomorrow is anything like today, the review will be have to be completed quickly before reviewers are pulled away for other tasks. Hence, this review must proceed using only a handful of examples from the local domain.

Some software engineering domains are amenable to big data solutions where large models are built from vast supplies of background knowledge. But Devanbu argues that software engineering, there are many phenomena that are known to be highly project-specific (e.g. the CPU or time required for one project's usual SQL query may be very different to another). Such project-specific data can be quite limited, especially early in the history of a project; thus is it useful to have models they can learn to perform a task with very few examples.

When speaking of large language models, Devanbu comments that a particularly exciting aspect of such learners is their knack for few-shot and zero-shot learning: they can learn to perform a task with very few examples. Few-shoting has particular synergies in software engineering, where there are a lot of phenomena that are known to be highly project-specific. However, project-specific data can be quite limited, especially early in the history of a project; thus the few-shot learning capacity of such learners might be very relevant.

Here we show that the same effect can be seen in other kinds of learners; specifically, the classifiers used within algorithms called tree-structured Parzen estimators (TPE) that explore semi-supervised multi-objective optimization. TPE is a natural model for supporting "time-constrained model-review"; i.e. groups of reviewers who have must perpetually review all the different software systems being used in the ever-evolving tool space that surrounds them. Given recent advances in large language models, and certain changes in the legislative environment around software (see next section), we antiipate taht such a review proces will become increasingly common

8 Features of Time-Constrained Review

Just to be clear, we are not saying that all problems can be resolved via few dozen examples. In ultra-safety critical applications, or when building generative models, or (outside of SE) when trying to find another 1% improvement in (say) the airflow over a wingtip going through the sound barrier, it is required to process as much information as possible.

Table 1 Please write your table caption here

first	second	third
number	number	number
number	number	number

8.1 Frequent reviews

8.2 Gaps between the reviews

9 multi-objective reasonong

10 Sem-spervised

11 Limited knowledvge of ssytem internals

Machine learning for busy people should not strive for (e.g.,) elaborate theories or (e.g.,) increasing the expressive power of the language of the theory. Rather, a better goal might be to find the smallest theory with the most impact.

My lesson from working on data sets from the PROMISE conference on rep

Your text comes here. Separate text sections with

12 Section title

Text with citations [**?**] and [**?**].

12.1 Subsection title

as required. Don't forget to give each section and subsection a unique label (see Sect. 12).

Paragraph headings Use paragraph headings as needed.

$$a^2 + b^2 = c^2 \tag{2}$$

References

1. Amazon just showed us that 'unbiased' algorithms can be inadvertently racist (2016). URL https://www.businessinsider.com/how-algorithms-can-be-racist-2016-4
2. Machine bias. www.propublica.org (2016). URL https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing
3. Study finds gender and skin-type bias in commercial artificial-intelligence systems (2018). URL http://news.mit.edu/2018/study-finds-gender-skin-type-bias-artificial-intelligence-systems-0212

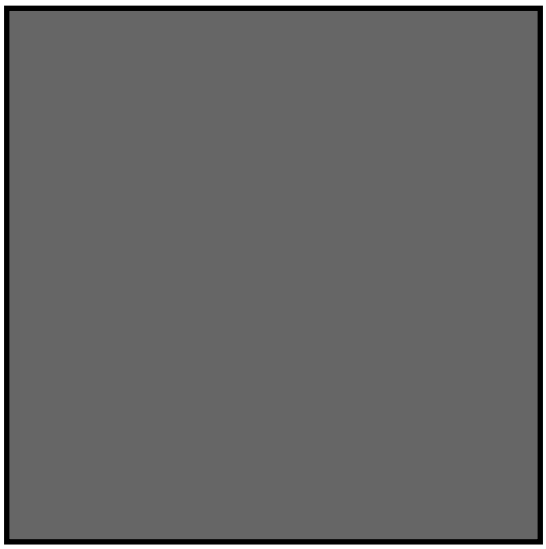

Fig. 1 Please write your figure caption here

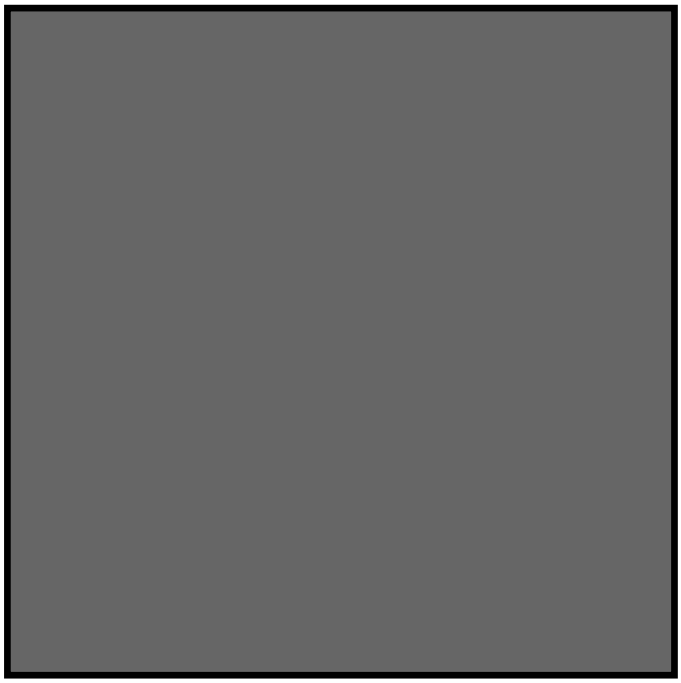

Fig. 2 Please write your figure caption here

4. Agrawal, A., Fu, W., Chen, D., Shen, X., Menzies, T.: How to "dodge" complex software analytics? arXiv preprint arXiv:1902.01838 (2019)
5. Aha, D.W., Kibler, D., Albert, M.K.: Instance-based learning algorithms. Mach. Learn. **6**(1), 37–66 (1991). DOI 10.1023/A:1022689900470. URL https://doi.org/10.1023/A:1022689900470
6. Ahmed, T., Devanbu, P.: Few-shot training llms for project-specific code-summarization. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ASE '22. Association for Computing Machinery, New York, NY, USA (2023). DOI 10.1145/3551349.3559555. URL https://doi-org.prox.lib.ncsu.edu/10.1145/3551349.3559555
7. Alhoshan, W., Zhao, L., Ferrari, A., Letsholo, K.J.: A zero-shot learning approach to classifying requirements: A preliminary study. In: International Working Conference on Requirements Engineering: Foundation for Software Quality, pp. 52–59. Springer (2022)
8. Alvarez, L., Menzies, T.: Don't lie to me: Avoiding malicious explanations with stealth. IEEE Software **40**(3), 43–53 (2023). DOI 10.1109/MS.2023.3244713
9. Amal, B., Kessentini, M., Bechikh, S., Dea, J., Said, L.B.: On the use of machine learning and search-based software engineering for ill-defined fitness function: a case study on software refactoring. In: International Symposium on Search Based Software Engineering, pp. 31–45. Springer (2014)
10. Amasaki, S.: Cross-version defect prediction: use historical data, cross-project data, or both? Empirical Software Engineering pp. 1–23 (2020)
11. Araújo, A.A., Paixao, M., Yeltsin, I., Dantas, A., Souza, J.: An architecture based on interactive optimization and machine learning applied to the next release problem. Automated Software Engineering **24**(3), 623–671 (2017)
12. Caliskan, A., Bryson, J.J., Narayanan, A.: Semantics derived automatically from language corpora contain human-like biases. Science **356**(6334), 183–186 (2017). DOI 10.1126/science.aal4230. URL https://science.sciencemag.org/content/356/6334/183
13. Catolino, G.: Just-in-time bug prediction in mobile applications: the domain matters! In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 201–202. IEEE (2017)
14. Chakraborty, J., Majumder, S., Menzies, T.: Bias in machine learning software: Why? how? what to do? In: FSE'21 (2021). DOI 10.1145/3468264.3468537
15. Chen, D., Stolee, K.T., Menzies, T.: Replication can improve prior results: A github study of pull request acceptance. CoRR **abs/1902.04060** (2019)
16. Chen, T.Y., Kuo, F.C., Liu, H., Poon, P.L., Towey, D., Tse, T., Zhou, Z.Q.: Metamorphic testing: A review of challenges and opportunities. ACM Computing Surveys (CSUR) **51**(1), 1–27 (2018)
17. Cruz, A.F., Saleiro, P., Belém, C., Soares, C., Bizarro, P.: Promoting fairness through hyperparameter optimization. In: 2021 IEEE International Conference on Data Mining (ICDM), pp. 1036–1041. IEEE (2021)
18. Easterby-Smith, M.: The design, analysis and interpretation of repertory grids. International Journal of Man-Machine Studies **13**(1), 3–24 (1980). DOI https://doi.org/10.1016/S0020-7373(80)80032-0. URL https://www.sciencedirect.com/science/article/pii/S0020737380800320
19. Faloutsos, C., Lin, K.I.: FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets, vol. 24. ACM (1995)
20. Gebru, T.: "race and gender
21. Green, B.: The flaws of policies requiring human oversight of government algorithms. Computer Law & Security Review **45**, 105681 (2022)
22. Herbold, S., Trautsch, A., Trautsch, F., Ledel, B.: Problems with szz and features: An empirical study of the state of practice of defect prediction data collection. Empirical Software Engineering **27**(2), 42 (2022)
23. Hindle, A., German, D.M., Holt, R.: What do large commits tell us?: a taxonomical study of large commits. In: Proceedings of the 2008 international working conference on Mining software repositories, pp. 99–108. ACM (2008)
24. Holte, R.C.: Very simple classification rules perform well on most commonly used datasets. Machine Learning **11**, 63–90 (1993). URL https://api.semanticscholar.org/CorpusID:6596

25. Johnson, B., Menzies, T.: Unfairness is everywhere. so what to do? an interview with jeanna matthews. IEEE Software (2023)
26. Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N.: A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering **39**(6), 757–773 (2012)
27. Kang, H.J., Aw, K.L., Lo, D.: Detecting false alarms from automatic static analysis tools: How far are we? In: Proceedings of the 44th International Conference on Software Engineering, ICSE '22, p. 698–709. Association for Computing Machinery, New York, NY, USA (2022). DOI 10.1145/3510003.3510214. URL https://doi.org/10.1145/3510003.3510214
28. Kim, S., Whitehead Jr, E.J., Zhang, Y.: Classifying software changes: Clean or buggy? IEEE Transactions on Software Engineering **34**(2), 181–196 (2008)
29. Kington, A.: Defining teachers' classroom relationships (2009)
30. Kohavi, R., John, G.H.: Wrappers for feature subset selection. Artificial Intelligence **97**(1-2), 273–324 (1997)
31. Le, V.H., Zhang, H.: Log parsing with prompt-based few-shot learning. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 2438–2449. IEEE (2023)
32. Lin, Y., Peng, X., Cai, Y., Dig, D., Zheng, D., Zhao, W.: Interactive and guided architectural refactoring with search-based recommendation. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 535–546 (2016)
33. Majumder, S., Chakraborty, J., Menzies, T.: When less is more: on the value of "co-training" for semi-supervised software defect predictors. Empirical Software Engineering **29**(2), 1–33 (2024)
34. McIntosh, S., Kamei, Y.: Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. IEEE Transactions on Software Engineering **44**(5), 412–428 (2017)
35. Menzies, T., Turhan, B., Bener, A., Gay, G., Cukic, B., Jiang, Y.: Implications of ceiling effects in defect predictors. In: Proceedings of the 4th international workshop on Predictor models in software engineering, pp. 47–54. ACM (2008)
36. Mockus, A., Votta, L.G.: Identifying reasons for software changes using historic databases. In: icsm, pp. 120–130 (2000)
37. Nam, J., Kim, S.: Clami: Defect prediction on unlabeled datasets. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015) (2015)
38. Noble, S.U.: Algorithms of oppression. New York University Press (2018)
39. Northpointe, I.: Practitioner's guide to compas core. (2015)
40. Pearson, K.: Principal components analysis. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science **6**(2), 559 (1901)
41. Platt, J.: Fastmap, metricmap, and landmark mds are all nyström algorithms. In: International Workshop on Artificial Intelligence and Statistics, pp. 261–268. PMLR (2005)
42. Rahman, F., Posnett, D., Herraiz, I., Devanbu, P.: Sample size vs. bias in defect prediction. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering, pp. 147–157. ACM (2013)
43. Rosa, G., Pascarella, L., Scalabrino, S., Tufano, R., Bavota, G., Lanza, M., Oliveto, R.: A comprehensive evaluation of szz variants through a developer-informed oracle. Journal of Systems and Software **202**, 111729 (2023). DOI https://doi.org/10.1016/j.jss.2023.111729. URL https://www.sciencedirect.com/science/article/pii/S0164121223001243
44. Rudin, C.: Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. Nature Machine Intelligence **1**(5), 206–215 (2019). DOI 10.1038/s42256-019-0048-x. URL https://doi.org/10.1038/s42256-019-0048-x
45. Shackelford, M.R.: Implementation issues for an interactive evolutionary computation system. In: Proceedings of the 9th annual conference companion on Genetic and evolutionary computation, pp. 2933–2936 (2007)
46. Shepperd, M., Song, Q., Sun, Z., Mair, C.: Data quality: Some comments on the nasa software defect datasets. Software Engineering, IEEE Transactions on **39**, 1208–1215 (2013). DOI 10.1109/TSE.2013.11

47. Simon, H.A.: Rational choice and the structure of the environment. Psychological Review **63**(2), 129–138 (1956)
48. Tu, H., Yu, Z., Menzies, T.: Better data labelling with emblem (and how that impacts defect prediction). IEEE Transactions on Software Engineering (2020)
49. Valerdi, R.: Heuristics for systems engineering cost estimation. IEEE Systems Journal **5**(1), 91–98 (2010)
50. Vasilescu, B.: Personnel communication at fse'18. Found. Softw. Eng (2018)
51. Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., Filkov, V.: Quality and productivity outcomes relating to continuous integration in github. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 805–816. ACM (2015)
52. Wu, X., Zheng, W., Chen, X., Zhao, Y., Yu, T., Mu, D.: Improving high-impact bug report prediction with combination of interactive machine learning and active learning. Information and Software Technology **133**, 106530 (2021). DOI https://doi.org/10.1016/j.infsof.2021.106530. URL https://www.sciencedirect.com/science/article/pii/S0950584921000185
53. Wu, X., Zheng, W., Xia, X., Lo, D.: Data quality matters: A case study on data label correctness for security bug report prediction. IEEE Transactions on Software Engineering **48**(7), 2541–2556 (2022). DOI 10.1109/TSE.2021.3063727
54. Xu, Z., Li, L., Yan, M., Liu, J., Luo, X., Grundy, J., Zhang, Y., Zhang, X.: A comprehensive comparative study of clustering-based unsupervised defect prediction models. Journal of Systems and Software **172**, 110862 (2021). DOI https://doi.org/10.1016/j.jss.2020.110862. URL https://www.sciencedirect.com/science/article/pii/S0164121220302521
55. Yang, Y., Webb, G.I.: A comparative study of discretization methods for naive-bayes classifiers. In: Proceedings of PKAW, vol. 2002 (2002)
56. Yu, Z., Fahid, F.M., Tu, H., Menzies, T.: Identifying self-admitted technical debts with jitterbug: A two-step approach. IEEE Transactions on Software Engineering **48**(5), 1676–1691 (2022)
57. Yu, Z., Menzies, T.: Fast2: An intelligent assistant for finding relevant papers. Expert Systems with Applications **120**, 57–71 (2019)