How to Get Your Paper Rejected!

(Bad Smells in the Software Science Papers)

Tim Menzies, Rahul Krishna
North Carolina State University, USA
tim@menzies.us,i.m.ralk@gmail.com

ABSTRACT

Listed here are some "bad smells" that decreases the odds of acceptance of a submitted paper in software science. While this list is subjective, the authors of this paper apply it dozens of times each year. Hence, for the researchers submitting papers that might be reviewed by us, this list could be quite useful.

This list is incomplete, and that is deliberate. The aim of this paper is to encourage more debate on what constitutes a "valid" paper. We hope that this sparks a debate that significantly matures this list.

KEYWORDS

Bad smells, Best practices, Defect prediction.

ACM Reference format:
Tim Menzies, Rahul Krishna . 2018. How to Get Your Paper Rejected!. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 4 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software science methods distill large amounts of low-value data into small chunks of very high-value information. For example, after examining many software projects, certain coding styles could be seen to be more bug prone (and should be avoided).

Each year, the author reviews many software science papers (the actual number varies but a common range is between 50 to 100). This paper lists the "bad smells" that increase the odds that these authors might reject a paper.

"Bad smells" is a term comes from the agile community. According to Fowler [4], bad smells (a.k.a. code smells) are "a surface indication that usually corresponds to a deeper problem". Just like Fowler, we say that the 'smells' of this paper do not necessarily indicate that a paper must be rejected. But, fixing it may significantly improve the odds of the paper being viewed favourably.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 BAD SMELLS

The following list is sorted in decreasing order of "smelliness". That is, early items usually lead to rejection while later items, somewhat less so.

2.1 Being Boring (the worst smell of all)

All science must strive to be correct and precise. Yet in this age of social media, and invitation-only journal-first papers, it is important that a paper is also interesting.

Papers should have a point. They should contribute to a current debate. Or, if no such debate exists, they should review the current literature and note a gap that requires debating. Note that, this review need not be as complex as a full systematic literature review (that can take months to complete [16]). For examples of such lightweight reviews, see [2] (specifically Figure 2, §2.4, and Table 3). Here, the researchers find a research gap in 22 papers of the top-most cited "defect prediction" papers that have at least 10 cites per year over the last decade.

2.2 Not Reading Related Work

The next two points should not exist. Nevertheless, we know of many papers that do not heed the following advice.

- Do not base you literature review on papers that are mostly ten years or older.
- If you are submitting to venue X, spend some time reviewing the recent papers from that venue.

2.3 Using Deprecated Data

It is not recommended to use:

- Very old data; e.g., the old PROMISE data from last-century NASA project (JM1, CM1, KC1, KC2, etc.
- Problematic data; e.g., data sets with know quality issues (which, again, includes the old PROMISE data from last-century NASA project [28]).
- Data from trivial problems; e.g. the SEIMANS test suite containing such trivial problems as triangle inequality.

On the other hand, tiny problems do have certain benefits. For example, we recommended to use tiny problems for the purposes of explaining and debugging algorithms, prior to attempting scale up to much larger problems. For example, small synthetic problems like ZDT, DTLZ [10], and WFG [20] are very useful to evaluate a meta-heuristic algorithm. For further details about other test problems, please refer to [20].

That said, it is *not* recommended that you try publishing results based on these small problems. In our experience, SE reviewers are more interested in results from more realistic sources, rather than results from synthetic problems like ZDT or the SEIMANS suite.

Rather, it is better to publish results on wide range of interesting problems taken from the SE literature. This is particularly true since there now exists many on-line repositories where researchers can access a wide range of data from a large number of projects (e.g. GitHub, GitTorrent[1]; and others [2]). Given the existence of such data sources, it is now normal to publish using data extracted from hundreds of projects (e.g. [3, 22]).

2.4 Not Exploring Stability

For many reasons, SE data sets often exhibit large variances [26]. So it is important to check if the effect you want to report actually exists, or is just due to noise. Hence:

- When multiple data sets are available, try many data sets.
- Report median and IQR (75th-25th percentile range).
- Do not average results across N data sets. Instead, use creative graphics to succinctly summarize the results across your data sets.e.g. see Figure 2 of [12].
- If exploring only one data set, then repeat the analysis (say) 20 to 30 times;

Note that "creative graphics" can include tables. Our rules for easy table reading are as follows:

- Show the treatment you want to endorse in the first or last column.
- For each row, highlight (with say) a background effect which treatment is statistically the "best".
- In the last row, report as a percentage, the number of times each treatment was the best.
- Reduce the number of significant figures in the table. E.g. when reporting precision, do not write "0.9176". Instead, "92" will suffice.

There are many ways to obtain a large enough sample (with 20-30 repeats) to perform the necessary statistical test:

- Do the whole thing many times on some 90% sample of the data, selected at random.
- Temporal validation: given data divided into some time series (e.g. different versions of the same project) then train on past data and test on future data. Our recommendation here is to keep the size of the future test set constant; e.g. train on software version K-2, K-1 then test on version K.
- Cross-project validation: train on another project data, then test on the project. Note that such cross-project transfer learning can get somewhat complex when the projects use data with different attribute names.
- Within-project validation (also called "cross-validation"): divide the project data into N bins, train on $N - 1$ bins, test on the hold out. Repeat for all bins.

Note that, temporal validation does not produce a range of results since the target is always fixed. However, cross- and within- produce distributions of results and so must be analyzed with appropriate statistics. As a generally accepted rule, in order to obtain, say, 25 samples of cross-project learning, apply the 90% sample method 25 times. And to obtain, say, 25 samples of within-project learning, $M = 5$ times shuffle the order of the data and for each shuffle, and run $N = 5$ cross-val. Caveat: for data sets under, say, 60 items, use $M, N = 8, 3$ (since $8 * 3 \approx 25$).

2.5 Using Statistics, Poorly

We strongly discourage reports of the form "we show a statistically significant improvement of 3%". We have read so many of those by now, it is rather disappointing to see that we have not achieved a 200% precision and recall.

Also, using "significance" (as it is precisely defined in the statistical community) as the sole tool to judge of the value of a result, is widely deprecated—especially by statisticians [6]. Such significance tests need to be augmented by an effect size test, lest we derive spurious conclusions [21].

A good statistical analysis is often required to pass peer review. But those statistics should be interpreted post-implementation (as a sanity check) and not the as the prime argument. So, to compare if one treatment is better than another, apply the following rules:

- Visualize the data, somehow.
- Check if the central tendency of one distribution is better than the other; e.g. compare their median values.
- Check if the distributions are significantly different;
- If so, verify that the difference between the central tendencies is not due to a small effect.

Here are some additional tips:

- Try to make your argument visually before resorting to reporting statistics. See [33] for a wonderful example of a very convincing visual argument.
- Since SE data sets can be highly skewed, we recommend performing a non-parametric reasoning about statistical significance (for example using a bootstrap test).
- Likewise for effect size. The standard non-parametric effect size test is the Cliffs Delta which runs in time $O(N^2)$. This can be reduced to $O(N \cdot log(N))$ by a pre-sort of the data[3].
- Since bootstrapping requires hundreds of samples of large data spaces, we recommend that bootstrapping is only executed after the effect size test has passed.
- Another way to reduce the number of calls to bootstrap is to recursively dichotomize the data, stopping whenever the effect size test or significance test report that further divisions are not statistically different. This recursive clustering is the Scott-Knott test.

Implementations of Scott-Knott that uses bootstrapping and Cliffs-Delta to control its recursive bi-clustering are available as open-source packages[4].

2.6 Not Exploring Simplicity

One interesting, and humbling, aspect of using Scott-Knott tests is that, often, dozens of treatments and grouped together into just a few ranks. For example, in the Scott-Knott results of [15], 32 learners were divided into only four groups. This means that (a) many learning methods produce equivalent results; (b) we should stop fussing about minor differences between data mining results since, in the grand scheme of things, the overall structure of the SE data

[1]https://github.com/cjb/GitTorrent
[2]http://tiny.cc/seacraft

[3]See https://gist.github.com/timm/a6e759eb7d9b5f05b468 for details.
[4]See https://goo.gl/uoj3FL

only supports a few conclusions; (c) SE data mining need not always be so hard.

For further evidence of the inherent simplicity of some SE tasks, see [11, 17, 19].

That said, it is surprising to report that in the SE literature there are very many complex solutions to very many problems. Researchers in empirical methods *strongly* advocate comparing a seemingly sophisticated method against some simpler alternative. Be aware that such comparisons can be very humbling. Often when we code and test the seemingly stupider and simpler method, we find it has some redeeming feature that makes us abandon the more complex methods [7, 12, 23].

This plea, to explore simplicity, is particularly important in the current research climate that is often focused on a very slow method called Deep Learning. Deep Learning is inherently better for problems that have highly intricate internal structure. That said, many SE data sets have very simpler regular structures, so very simple techniques can execute much faster and perform just as well, or better, than Deep Learning. Such faster executions are very useful since they enable easier replication of prior results.

This is not to say that Deep Learning is always useless. Rather, it means that any Deep Learning result should also be compared against some simpler baseline (e.g. [12]) to justify the additional complexity.

Three simple methods for exploring simpler options are *scouts*, *stochastic search* and *feature selection*. Holte reports that a very simple learner called 1R can be *scout* ahead in a data set and report back if more complex learning will be required. More generally, by first running a very simple algorithm, it is easy to quickly obtain baseline results to glean the overall structure of the problem.

As to stochastic search, for optimization, we have often found that *stochastic searches* (e.g. using differential evolution [29]) performs very well for a wide range of problems [1, 2, 13].

Note that Holte might say that an initial stochastic search of a problem is a *scout* of that problem.

Also, for classification, we have found that *feature selector* will often find most of the attributes are spurious and can be discarded [5]. This is important since models learned from fewer features are easier to read, understand, critique, and explain to business users. Also, by removing spurious features, there is less chance of researchers reporting a spurious conclusion. Further, after reducing the dimensionality of the data, then any subsequent processing is faster and much easier.

Note that many papers perform a very simple linear-time feature selection; e.g. (i) sort by correlation, variance, or entropy; (ii) then remove the worst remaining feature; (iii) build a model from the surviving features; (iv) stop if the new model is worse than the model learned before; (v) otherwise, go back to step (ii). Note that this is deprecated. Hall & Holmes report that such greedy searchers can stop prematurely and that better results are obtained by growing in parallel sets of useful features (see their CFS algorithm, described in [16]).

[5] For example, the feature selection experiments of [25] show that up to 39 of 41 static code attributes can be discarded while preserving the predictive prowess. A similar result with effort estimation, where most of the features can be safely ignored is reported in [8]

Before moving on, we note one paradoxical aspect of simplicity research—it can be very hard to do. Before certifying that some simpler method is better than the most complex state-of-the-art alternative, it may be required to test the simpler against the complex. This can lead to the peculiar situation where weeks of CPU are required to evaluate the value of (say) some very simple stochastic method that takes just a few minutes to terminate.

2.7 Not Justifying Choice of Learner

When certifying some new method, researchers need to compare their methods across a range of interesting algorithms.

For learners that predict for discrete class learning, it useful to see Table 9 of [15]. This is a clustering of 32 learners commonly used for defect prediction (presented at ICSE'15). The results clusters into four groups, best, next, next, worst. We would recommend:

- Selecting your range of comparison algorithms as one (selected at random) for each group;
- Or selecting your comparison algorithms all from the top group.

For learners that predict for continuous classes, it is not clear what is the canonical set. That said, Random Forest regression trees and logistic regression are widely used.

For search-based SE problems, we recommend an experimentation technique called (you+two+next+dumb), defined as

- "You" is your new method;
- "Two" are well-established widely-used methods (often NSGA-II and SPEA2 [27]);
- A "next" generation method e.g. MOEA/D [34], NSGA-III [9];
- and one "dumb" baseline method (random search or SWAY [7]).

2.8 Not Tuning

Many software systems, including data miners, have poorly chosen defaults [1, 2, 13, 18, 30, 32].

Numerous recent results show that the default control settings for data miners are less than optimum. For example, when performing defect prediction, various groups report that tuning can find new settings that dramatically improve the performance of the learned mode [13, 30, 31]. Also, when using SMOTE to re-balance data classes, tuning found that for distance calculations using

$$d(x, y) = \left(\sum_i (x_i - y_i)^n \right)^{1/n}$$

the Euclidean distance of $n = 2$ usually works far worse than another distance measure using $n = 3$.

As to how to tune, we *strongly* recommend against the "grid search" used by Lessmann et al. [24] or Tantithamthavorn et al. [31]. ; i.e. a set of nested for-loops that try a wide range of settings for all tuneable parameters. This is a slow approach—we have explored grid search for defect prediction and found it takes days to terminate [13]. Not only that, we found that grid search can miss important optimizations [14]. Every grid has "gaps" between each grid division which means that a supposedly rigorous grid search can still miss important configurations [5]. Bergstra and Bengio [5] comment that for most data sets only a few of the tuning parameters really matter– which means that much of the runtime associated with grid search is actually wasted. Worse still, Bergstra and Bengio comment that the important tunings are different for different data sets—a phenomenon that makes grid search a poor choice for configuring data mining algorithms for new data sets.

Rather than using "grid search", we have found that very simple optimizers (such as differential evolution [29]) suffice for finding better tunings. Such optimizers are very easy to code from scratch. They are also readily available in open-source toolkits such as jMETAL[6] or DEAP[7].

3 DISCUSSION

This list is not exhaustive by any means, and this is deliberate. For example, this paper does not discuss the complex issue of replication packages. In the very near future, we foresee a time when it will become very rare to publish *unless* the results are reproducible in some replication package. We can report here that, at least for quantitative software science based on data mining algorithms, such reproducibility is now common practice. In fact, many conferences in software engineering reward researchers with "badges", if they place their materials online. Another issue we have not considered here is assessing the implication of using different criteria; or studying the generality of methods for software analytics. Recent studies [23] have shown that seemingly state-of-the-art techniques for tasks such as transfer learning that were designed for one domain (defect prediction) in fact translate very poorly when applied to other domains (such as code-smell detection, etc.).

That said, the aim of this paper is to encourage more debate on what constitutes a "valid" paper. We hope that a further, more considered, debate significantly matures this list.

REFERENCES

[1] Amritanshu Agrawal, Wei Fu, and Tim Menzies. 2016. What is Wrong with Topic Modeling? (and How to Fix it Using Search-based SE). *CoRR* abs/1608.08176 (2016). arXiv:1608.08176 http://arxiv.org/abs/1608.08176

[2] Amritanshu Agrawal and Tim Menzies. 2017. "Better Data" is Better than "Better Data Miners" (Benefits of Tuning SMOTE for Defect Prediction). *CoRR* abs/1705.03697 (2017). arXiv:1705.03697 http://arxiv.org/abs/1705.03697

[3] Amritanshu Agrawal, Akond Rahman, Rahul Krishna, Alexander Sobran, and Tim Menzies. 2017. We Don't Need Another Hero? The Impact of "Heroes" on Software Development. *CoRR* abs/1710.09055 (2017). arXiv:1710.09055 http://arxiv.org/abs/1710.09055

[4] Kent Beck, Martin Fowler, and Grandma Beck. 1999. Bad smells in code. *Refactoring: Improving the design of existing code* (1999), 75–88.

[5] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13, Feb (2012), 281–305.

[6] Ronald P Carver. 1993. The case against statistical significance testing, revisited. *The Journal of Experimental Education* 61, 4 (1993), 287–292.

[7] Jianfeng Chen, Vivek Nair, Rahul Krishna, and Tim Menzies. 2018. " Sampling" as a Baseline Optimizer for Search-based Software Engineering. *IEEE Transactions on Software Engineering* (2018).

[8] Zhihao Chen, Tim Menzies, Daniel Port, and D Boehm. 2005. Finding the right data for software cost modeling. *IEEE software* 22, 6 (2005), 38–46.

[9] Kalyanmoy Deb and Himanshu Jain. 2014. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints. *IEEE Trans. Evolutionary Computation* 18, 4 (2014), 577–601.

[10] Kalyanmoy Deb, Lothar Thiele, Marco Laumanns, and Eckart Zitzler. 2005. Scalable test problems for evolutionary multiobjective optimization. *Evolutionary Multiobjective Optimization. Theoretical Advances and Applications* (2005).

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

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

[13] Wei Fu, Tim Menzies, and Xipeng Shen. 2016. Tuning for Software Analytics. *Inf. Softw. Technol.* 76, C (Aug. 2016), 135–146. https://doi.org/10.1016/j.infsof.2016.04.017

[14] Wei Fu, Vivek Nair, and Tim Menzies. 2016. Why is Differential Evolution Better than Grid Search for Tuning Defect Predictors? *CoRR* abs/1609.02613 (2016). arXiv:1609.02613 http://arxiv.org/abs/1609.02613

[15] Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. 2015. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 789–800.

[16] Mark A Hall and Geoffrey Holmes. 2003. Benchmarking attribute selection techniques for discrete class data mining. *IEEE transactions on knowledge and data engineering* 15, 6 (2003), 1437–1447.

[17] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 763–773.

[18] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A Self-tuning System for Big Data Analytics.. In *Cidr*, Vol. 11. 261–272.

[19] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 837–847.

[20] Simon Huband, Philip Hingston, Luigi Barone, and Lyndon While. 2006. A review of multiobjective test problems and a scalable test problem toolkit. *IEEE Transactions on Evolutionary Computation* (2006).

[21] Ekrem Kocaguneli, Thomas Zimmermann, Christian Bird, Nachiappan Nagappan, and Tim Menzies. 2013. Distributed development considered harmful?. In *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 882–890.

[22] Rahul Krishna, Amritanshu Agrawal, Akond Rahman, Alexander Sobran, and Tim Menzies. 2017. What is the Connection Between Issues, Bugs, and Enhancements? (Lessons Learned from 800+ Software Projects). *CoRR* abs/1710.08736 (2017). arXiv:1710.08736 http://arxiv.org/abs/1710.08736

[23] Rahul Krishna and Tim Menzies. 2017. Simpler Transfer Learning (Using "Bellwethers"). *arXiv* abs/1703.06218 (2017). http://arxiv.org/abs/1703.06218

[24] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Trans. Softw. Eng.* 34, 4 (July 2008), 485–496. https://doi.org/10.1109/TSE.2008.35

[25] Tim Menzies, Jeremy Greenwald, and Art Frank. 2007. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering* 33, 1 (2007), 2–13.

[26] Tim Menzies and Martin Shepperd. 2012. Special issue on repeatable results in software engineering prediction. (2012).

[27] Abdel Salam Sayyad and Hany Ammar. 2013. Pareto-optimal search-based software engineering (POSBSE): A literature survey. In *Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2013 2nd International Workshop on*. IEEE, 21–27.

[28] Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. 2013. Data quality: Some comments on the nasa software defect datasets. *IEEE Transactions on Software Engineering* 39, 9 (2013), 1208–1215.

[29] Rainer Storn and Kenneth Price. 1997. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization* 11, 4 (1997), 341–359.

[30] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 321–332.

[31] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2018. The Impact of Automated Parameter Optimization on Defect Prediction Models. *IEEE Transactions on Software Engineering* (2018).

[32] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1009–1024.

[33] Leland Wilkinson, Anushka Anand, and Dang Nhon Tuan. 2011. CHIRP: a new classifier based on composite hypercubes on iterated random projections. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 6–14.

[34] Qingfu Zhang and Hui Li. 2007. MOEA/D: A multiobjective evolutionary algorithm based on decomposition. *IEEE Transactions on evolutionary computation* 11, 6 (2007), 712–731.

[6]jmetal.sourceforge.net/
[7]github.com/DEAP/deap