# Seven Principles of Inductive Software Engineering: What we do IS Different.

Tim Menzies

## Different and Important

*Inductive software engineering* is that branch of software engineering focusing on the delivery of data mining based software applications to users. Within those data miners, the core problem is *induction* which is the extraction of small patterns from larger data sets. Inductive engineers spend much effort trying to understand user goals in order to inductively generate the models that most matter to those user.

Previously, with Christian Bird, Thomas Zimmermann, Wolfram Schulte, and Ekrem Kocaganeli, we wrote an *Inductive Engineering Manifesto* [1] that offered some details on this new kind of engineering. That whole manifesto is a little long, so here I offer a quick summary. Listed below are seven key principles which, if ignored, can make it harder to deploya analytics in the real world. For more details (and more principles), the reader should refer to the original document [1].

## Principle #1: Users Before Algorithms

Mining algorithms are only good if users find their use in real-world applications. This means that they need to

* understand the results
* understand that those results add value to their work.

Accordingly, it is strongly recommend that once the algorithms generate some model, then the inductive engineer *talks to the users* about those results.

In our experience, such discussions lead to a second, third, fourth, etc round of learning. To assess if you are talking in “the right way” to your users, check the following:

* Do users bring their senior management to the meetings? If yes, great!
* Do users keep interrupting (you or each other) and debating your results? If yes, then shut the hell up (and take lots notes!)
* Do the users indicate they understand your explanation of the results? E.g. can they correctly extend your results to list desirable, undesirable implications of your results?
* Do your results are touching on issues that concern them? This is *easy* to check…. just count how many times they glance up from their notes, looking startled or alarmed.
* Do your user offer more data sources for analysis? If yes, they like what you are doing and want you to do it more.
* Do users invite you to their workspace and ask you to teach them how to do XYZ? If yes, this is a real win. Ideally, inductive software engineers build a team of engineers and users, each of which can contribute significantly to the process.

## Principle #2: Plan for Scale

Data mining methods are usually repeated multiple times in order to:

* Answer new user questions, inspired by the current results;
* Enhance data mining method or fix some bugs;
* Deploy the results, or the analysis methods, to different user groups

So that means that, if it works, you will be asked to do it again (and again (and again (and again))). To put that another way *thou shalt not click*. That is, if all your analysis requires lots of point-and-clock in a pretty GUI environment, then you are definitely **not** planning for scale.

Another issues is that as you scale up, your methods will need to scale up as well. For example, in our *Manifesto* document, we discussed the CRANE project at Microsoft that deployed data mining methods to the code base of Microsoft Windows. This was a *very large* project so the way it started was *not* the way it ended:

* Initially, a single inductive engineer did some rapid prototyping for a few week, to explore a range of hypotheses and gain user interest (and get user feedback on the early results);
* Next, the inductive engineering team spent a few months conducted many experiments to find stable models (and to narrow in on the most important user goals).
* In the final stage, which took a year, the inductive engineers integrate the models into a deployment framework that was suitable for target user base

Note that the team size doubled at each stage– so anyone funding this works needs to know that increasingly useful conclusions can be increasingly expensive.

## Principle #3: Get Early Feedback

This mentioned this above, but it is worth repeating. Before conducting very elaborate studies (that take a long time to reach a conclusion), try applying very simple tools to gain rapid early feedback

So simplicity first! Get feedback early and often! For example, there are many linear time discretizers for learning what are good divisions of continuous attributes (e.g. the Fayyad-Irani discretizer [2]). These methods can also report when breaking up an attribute is *not* useful since that attribute is not very informative. Using tools like these, it is possible to discover what attributes can be safely ignored (hint: usually, its more than half).

## Principle #4: Be Open-Minded

The goal of inductive engineering for SE is to find better ideas than what were available when you started. So if you leave a data mining projects with the same beliefs as when you’ve started, you really wasted a lot of time and effort. Hence, some mantras to chant while data mining are:

* Avoid a fixed hypothesis. Be respectful but doubtful to all user-suggested domain hypotheses. Certainly, explore the issues that they raise but also take the time to look further afield.
* Avoid a fixed approach for data mining (e.g. just using decision trees all the time), particularly for data not been mined before
* The most important initial results are the ones that radically and dramatically improve the goals of the project. So seek important results.

## Principle #5: Be Smart with your Learning

Lets face it, any inductive agents (human or otherwise) has biases that can confuse the learning process. So don’t torture the data to meet preconceptions (that is, it is ok to go “fishing” to look for new insights).

It also true that any inductive agent (and this includes you) can make mistakes. If organizations are going to use your results to change policy, the important outcomes are riding on your conclusions. This means you need to check and validate your results:

* Ask other people to do code reviews of your scripts.
* Check conclusion stability against different samples policies. For example:
* Policy1: divide data into (say) ten 90% samples (built at random). How much do your conclusions change across those samples?
* Policy2: sort data by collection date (if available). Learn from the far fast, the nearer past, then most recent data. Does time change your results? Is time *still* changing your results? It is important to check.
* Policy3,4,5,etc: Are there any other natural divisions of your data (e.g. east coast vs west coast; men vs women; etc)? Do they effect your conclusions?
* When reporting multiple runs of a learner, don’t just report mean results– also report the wriggle around the mean.
* In fact, do not report mean results at all since outliers can distort those mean values. Instead, try to report median and IQR results (the inter-quartile range is the difference between the 75th and 25th percentile).

## Principle #6: Live with the data you have

In practice, it is a rare analaytics projec that can dictate how data is collected in industrial contexts. Usually, inductive engieners have to cope with whatever data is available (rather than demand more data collected under more ideal conditions). Ths means that often you have to go mining with the data you have (and not the data you hope to have a some later date). So its important to spend some time on data quality operators. For example:

* Use *feature selection* to remove spurious attributes. There are many ways to perform such feature selection including the Fayyad Irani method discussed above. For a discussion of other feature selection methods, see [3].
* Use *row selection* to remove outliers and group related rows into sets of clusters. For a discussion on row selection methods, see [4]

One benefit of replaces rows with clusters is that any signal that is spread out amongst the rows can be “amplified” in the clusters. Ifwe cluster, then learn one model per cluster, then the resulting predictions have better median values and smaller variance [5].

In any case, we’ve often found row and feature selection discards up to 80 to 90% of the data, without damaging our ability to learn from the data. This means that ensuring quality of *all* the data can sometimes be less important that being able to extract quality data from large examples.

## Principle #7: Develop a Broad Skill Set that uses a Big Toolkit.

The reason organizations need to hire inductive engineers is that they come equiped with a very broad range of tools. This is important since many problems need specialized methods to find good solutions.

So, to become an inductive engineer, look for the “big ecology” toolkits where lots of developers are constantly trying out new ideas. Languages like Python, Scala (and lately, Julia) have extensive on-line forums were developers share their data mining tips. Toolkits like R, MATLAB, WEKA are continually being updated with new tools.

What a great time to be an inductive engineer! So much to learn, so much available to try. Would you want to have it any other way?

## References

1. Tim Menzies, Christian Bird, Thomas Zimmermann, Wolfram Schulte, and Ekrem Kocaganeli. 2011. The inductive software engineering manifesto: principles for industrial data mining. In Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering (MALETS ’11). ACM, New York, NY, USA, 19-26. DOI=http://dx.doi.org/10.1145/2070821.2070824
2. Usama M. Fayyad and Keki B. Irani. 1992. On the Handling of Continuous-Valued Attributes in Decision Tree Generation. Mach. Learn. 8, 1 (January 1992), 87-102. DOI=http://dx.doi.org/10.1023/A:1022638503176
3. Mark A. Hall and Geoffrey Holmes. 2003. Benchmarking Attribute Selection Techniques for Discrete Class Data Mining. IEEE Trans. on Knowl. and Data Eng. 15, 6 (November 2003), 1437-1447. DOI=http://dx.doi.org/10.1109/TKDE.2003.1245283
4. Olvera-López, J.Arturo; Carrasco-Ochoa, J.Ariel; Martínez-Trinidad, J.Francisco; Kittler, Josef. 2010. A review of instance selection methods Artificial Intelligence Review 34(2) pages 133-143 DOI=http://dx.doi.org/10.1007/s10462-010-9165-y
5. Menzies, T.; Butcher, A.; Cok, D.; Marcus, A.; Layman, L.; Shull, F.; Turhan, B.; Zimmermann, T., “Local versus Global Lessons for Defect Prediction and Effort Estimation,” in Software Engineering, IEEE Transactions on , vol.39, no.6, pp.822-834, June 2013 doi: http://dx.doi.org/10.1109/TSE.2012.83