White Paper

Defensive Data Science: What We Can Learn From Software Engineers

by David Marx

Data Scientist

August 11, 2015

ELDER RESEARCH
DATA SCIENCE & PREDICTIVE ANALYTICS

Defensive Data Science: What We Can Learn From Software Engineers

Table of Contents

Introduction

The Data Scientist’s skill set is a collection of coding and analytic talents, usually emphasizing predictive analytics knowledge over coding acumen. Josh Wills of Cloudera characterizes Data Scientists as people who are, “Better at coding than the average statistician,” which is probably accurate but is also setting the programming bar fairly low :-). Software engineers put a lot of thought and energy into the minutiae of their projects, like code structure and organization, and these considerations are a major contributing factor to the success of such projects. Predictive analytics projects have similar pain points as software development projects (with some additional aggravations unique to analytics), and so the data science community can gain significant benefit by adopting strategies and philosophies pioneered in the world of software engineering. Five such strategies — version control, code readability, documentation, semantic folder structure, and pipelines – are encouraged here.

Version Control

One of the simplest and most valuable practices that can be adopted by an analytics team is *version control*. Tools such as Git (http://git-scm.com) facilitate collaboration and make it significantly easier to recover from problems introduced by changes to code. The software development community implemented this tool decades ago, but it has only recently begun to gain traction in analytics and research communities. In addition to tracking changes to code, version control can also track small data sets, allowing you to associate data with each snapshot of your project. This extremely powerful strategy allows you to reproduce analytical experiments and audit how data and code changes have impacted your model or its results. Version control should be applied at the level of the project wherever possible, and the project or data product should live under its own repository. For most projects, a centralized repository should serve as the canonical codebase. Code in this repository should not be modified directly; instead, individual team members should work with local development copies of the project and update the centralized canonical codebase by “pushing” their changes (commits) to it. Segregating team member workspaces like this significantly reduces the risk of team members “stepping on each other’s toes.” Branching can be leveraged further to prevent analytics experiments, new features, or large structural changes from conflicting with other work in the project. An excellent overview of several common Git workflows is available on the Atlassian blog: https://www.atlassian.com/git/tutorials/comparing-workflows

Code Readability

Version control systems force a kind of documentation via commit messages, which provide short summaries of the changes associated with a commit. As informative as these may be, however, it is critically important that the code itself be descriptive and well documented. In-line comments in code can be helpful, but the best code is self-explanatory and requires few comments to understand, favoring meaningful variable and function names over reliance on in-line comments. It is common for scientific programmers to liberally use extremely compact abbreviations for variable names, but this practice often leads to code that is difficult for anyone but the author to interpret, and the authors themselves may not even be able to readily

understand their own work in the future. An excellent discussion of the importance and nuance of variable name selection can be found in chapter 11 of Steve McConnell's book Code *Complete*.
Intelligent use of whitespace is another way to increase code readability. The python language places a strong emphasis on code readability by imparting syntactic meaning to white space (among other clever design decisions); consequently, python code often reads almost like normal English. Furthermore, objects in python are self-documenting via built-in tools, like the dir() and help() commands, that allow a programmer to investigate the various attributes of an object that describe how it can or should be used.

Documentation

Self-explanatory code is always the goal, but even the clearest code should be accompanied by solid documentation. Documentation for functions should include descriptions of what they do, what kinds of inputs they take, what the output will look like, and perhaps some demo usage. The documentation associated with R packages on CRAN serves as an excellent example. Additionally, code files of all kinds (stand-alone functions, scripts, libraries, etc.) should include a "header" providing the "what/who/when/why/how" of the code in clearly identified sections:

- Summary (what does this do?)
- Author (Who wrote it?)
- Date (When was this first authored?)
- Purpose/Motivation (Why was this built?)
- Usage/Demo (How do I use this?)

Including this level of documentation takes some discipline, but the time taken to write a short note immediately after putting code on paper will pay for itself many times over in the future when team members can just read the documentation instead of reviewing someone else's code line-by-line to figure out how to use it.

Semantic Folder Structure

The "self-explanatory code" philosophy can be extended to the global project structure. A powerful strategy is to adopt a semantic folder structure in which the path/to/a/file encodes meaning. There should be a clear location to which any particular artifact associated with the project should go. Objects stored this way become very easy to find, and the location of objects disambiguates their purpose in the project. Example paths might be: *./code/modeling /train_ensemble.py*or *./data/raw/transactions.csv;* the "." denotes the location of the project folder (NB: this notation is referred to as a "relative path" in contrast with an "absolute path," which gives the explicit location starting with the drive name or root directory).

Pipelines

With a semantic folder structure in place, it should be fairly clear what purpose different files serve in the project. Even so, their inter-dependencies may not be obvious. The sequence in which different scripts need to be executed should be encoded explicitly in an executable pipeline that runs the project from front to back, including data ingestion, data transformation, model training, ensembling, scoring, and evaluation (for production models, there should be a separate pipeline just for scoring). For example, SAS Enterprise Miner (EM) provides a GUI for building what it calls "Process Flows." To remove ambiguity, it's preferable to encode this pipeline in a physical code file outside the GUI, which SAS EM is capable of generating. There are several open source solutions to assist in constructing these types of pipelines. The classic dependency management tool is GNU Make (http://gnu.org/software/make), which remains one of the best tools available today. One of the benefits of Make is that it has been in use for a long time, so there are many excellent tutorials for it (including several targeted towards analytics projects and researchers). If Make doesn't suit your needs, a popular, more full-featured free tool is Spotify's Luigi (http://luigi.readthedocs.org). Luigi is a python package for building and handling pipelines. Some of Luigi's benefits over Make include a visualization tool, a scheduler, and more granular failure management. In the absence of a dedicated tool, a pipeline can simply be constructed from a master script that runs everything in the appropriate sequence. Modularize such a file to allow certain parts of the pipeline to be turned on or off as needed. Pipelines constructed from tools like Make generally won't need this kind of modularity since the tool will automatically check to see if there have been modifications to upstream data files and only run the downstream code if necessary.

Conclusion

Data science is an experimental and creative science that often depends heavily on software development, but many analytics teams have yet to incorporate several valuable techniques pioneered in the software engineering world. Some of the most valuable of these, summarized here, are version control, readable code, liberal documentation, semantic folder structure and project organization, and pipelining tools. Using these strategies in analytics projects requires some overhead but can significantly streamline development in the near term and avoid a lot of headaches in the long term. Borrow these proactive engineering strategies to achieve long-term analytic success.

About the Author

David Marx is a Data Scientist at Elder Research, a consulting firm specializing in predictive analytics with offices in Arlington, Charlottesville and Baltimore. Before joining the Elder Research team, David earned a Master of Science in Mathematics and Statistics from Georgetown University and worked as a Senior Data Analyst/Software Developer at SoundExchange, a company that processes and distributes royalties for the music industry. His expertise includes statistical modeling, anomaly detection, natural language processing, recommendation systems, and graph analytics.

www.elderresearch.com

ELDER RESEARCH
DATA SCIENCE & PREDICTIVE ANALYTICS

National Capital Region	**Headquarters**	**Maryland Office**
2101 Wilson Boulevard	300 W. Main Street	839 Elkridge Landing
Suite 900	Suite 301	Suite 215
Arlington, VA 22201	Charlottesville, VA 22903	Linthicum, MD 21090
855.973.7673	434.973.7673	855.973.7673