Colin Lockard
Userid: lockardc
CSE 503
3 April 2017

HW1: I Have So Many Problems

Problem 1: A missing framework for machine learning experiments

I write systems that attempt to extract information from natural language text. Like many programmers today, I am using machine learning techniques and spend a lot of time running experiments in which I am tweaking various settings, from hyperparameters to featurization code to the actual data that I am running on. A major problem I have is accurately keeping track of the experimental setup and corresponding results of each experiment.

For example, let's say that last week I ran some experiments testing various regularization settings, and I'm now looking back at those results. I probably assigned an informative name to the folders and files containing the results, but there will be things I'm not sure of: Did I run these before I fixed that bug in the data preprocessing? Was this before or after I switched the featurization code to make all the text lowercase? Was I using the new, updated version of the dataset yet? There are other, more nefarious things that can cause problems as well: Do I have bug that is causing datapoints from my test set to pollute my training data?

These are problems that lie somewhere at the nexus of version control, testing, and data presentation/visualization. I can imagine a framework or language for running experiments that would both assist a programmer in dealing with these issues and also provide a more standardized way of recording and presenting experiments that could help with the reproducibility of published results.

For example, I can imagine a type system that has separate types for TrainingDataPoints and TestDataPoints that would throw an error if you tried to mix them, or even would check to confirm that no identical datapoints existed with both types. This framework could have a special type of abstract function called a Featurizer that takes as input a DataPoint and outputs a single Feature for that example. Individual Features can be composed into FeatureVectors, and Featurizers could be composed to create high-level features from lower-level features. A Featurized dataset could then be "compiled" into a training/test set. Some sort of version control would be tracking the Featurizers and DataPoints used to easily diff between different experiments, with similar attributes for model parameters.

Current version control systems don't quite seem suited--it doesn't make sense to commit to Git every time you change a hyperparameter. A bit of search has turned up a few attempts at frameworks to solve this, of which this one looks the most relevant. However, while this help track changes in parameters, I don't think it tracks changes in actual code. I think that some sort of DSL that helps us think about running experiments based on data the way SQL helps us with storing/accessing data would have a lot of applications today.

Problem 2: Understanding whether bad results are due to bug or technique

The scenario of writing research software also complicates the testing process. I imagine that most software developers have a very clear idea of the intended behavior of their code, so in general, in addition to writing unit tests to test various individual components, they also have the high-level test of the ultimate behavior of the entire program: Setting aside corner cases, if I run it, does it generally seem to be doing what I intended?. When writing research code, the researcher doesn't necessarily know how the program "should" be behaving. If I am testing a new machine learning model and it is producing terrible output, is that because the code is all correctly written but my model is just ineffective, or is it because I have a bug somewhere that is preventing the model from doing what it is capable of doing?

The basic idea of a "sanity check" is much harder in this setting. Good unit testing is still a viable option in this regime (though I suspect that it is rarely used by most researchers). I also think this paradigm supports slightly different testing practices than typical software development. I suspect that researchers are generally less concerned about ensuring that the program will never crash in unlikely edge cases than in generally having functional code.

Perhaps this is as simple as mocking out the "experimental" part of the code, such as machine learning model. But this portion of the code often consists of multiple moving parts, any one of which might have its own bugs. Perhaps there is simply some way to intelligently design a few test cases that can provide known results.

Problem 3:

One issue I have is related to maintenance, in particular, the potential change over time of the kind of data being input into the program. For example, my code often relies on external libraries for dealing with certain aspects of data. In working with natural language text, I use a package called Stanford CoreNLP to perform low-level NLP tasks on raw text. I also make use of a tool called WordNet which does things like provide synonyms for a word. Sometimes revisions to these tools have major ramifications on my downstream code; they don't necessarily break things in a way that throws exceptions, but they alter the data in such a way that it produces a different featurization that causes my machine learning models to produce poor results.

In addition, when running machine learning models in the real world, sometimes data comes in that looks very different than what my models were trained one; perhaps I trained on "clean" data but am now running on text scraped from the web that has stray bits of HTML in it.

It would be nice to have some kind of "data sanity check" that can detect that the inputs to the program have changed over time. I'm thinking that it would almost be like a fuzzier version of a type system. The program would learn a "type" for the input based on various features, and would throw some kind of exception if a data point didn't fit that type. As a very simple example, rather than simply restricting an input to be of Float type, it could learn that input is generally a positive float that is less than one million, and if a value outside that range appeared one day, it would alert the programmer/user that something unusual had happened.