CSE 503 Homework 1

Beibin Li (ID: 1723402)

Jan/7/2018

I had visa issue and absent the first two classes; so, I spent some extra time on doing this homework (4 hours on reading lecture notes, 5 hours on doing the homework).

Reuse Code

Reusing code is beneficial: (1) when you make a change to a function, you just need to make the change in one place rather than many places. (2) Reusing code can also avoid writing bugs while reimplementing the same functionality. (3) Reusing code can improve the readability and reduce complexity of code.

However, reuse other's code is hard unless the code is well documented. Unfortunately, most code in CS research are not well-documented or well-commented because of time limitation: researchers have to spent time on writing grants, writing papers, presenting projects, and attending conferences. Even if a program might be well-described in published papers, its code can be hard to understand due to lack of comments and undocumented assumptions. It's also hard to locate a function/module in a large project (e.g. more than 100k lines of code) if you are not familiar enough to the project. Sometimes, you do not even know a functionality is already implemented by others in the past.

Sometimes programmers can hardly reuse their own code. Often, programmers have to try different function calls for a single line of code (particularly true for scripting languages), and they usually choose to just comment redundant code out (or even leave it as is) rather than deleting it for reproducibility (e.g. to reproduce testing accuracy, plots, etc). However, if programmer doesn't delete duplicate or "dirty" code, the script would become long, unreadable, and not efficient to run.

If code is well-documented, reusability would be largely increased. One solution to the problem is to create "markdown" or "html" version of the code (e.g. R Markdown, iPython Notebook) so that comments can be easily made and viewed. However, installing and using these tools might be time consuming for programmers (even if it just take few minutes), and those systems usually require programmers to run code in browser. Another possible solution is to create better "comment system" in IDEs (including Vim and Emacs), which can display comments nicely and show LaTeX formulas. So, programmers do not need to install new software nor use browser to view/edit codes.

From my personal experience, creating UML graph or logic graph would increase readability of code significantly. By reading a UML graph, new teammates (labmates, grad students) can understand large projects easily, and they can find bugs and reusable codes much quicker. However, creating and maintaining these graphs are not easy, and PIs usually do not create such documentation because it's not helpful to get funding. A better UML ploting software (as a plug-in in Comment System or Markdown language) can be useful.

Break large component of code to smaller pieces might be a solution to reuse code, because large component of code often have assumptions and hence hard to be reused. However, breaking large component is time consuming. A possible alternative solution is to create tools to break a large function into smaller ones. Sometimes, programmer can hardly break a large component because there are too many assumptions made in the code, and I often encountered this problem while adding features or performing maintenance.

Test Suites

I have created and used test cases while creating software and video games, but I rarely created test cases in research (for data analysis). However, creating test cases can ensure soundness and robustness for research.

I encountered the following problems while creating test cases for my data analysis codes: (1) Create test cases by hand is almost impossible in many domains. e.g. create brain fMRI signal data, which contains millions lines of data for one

experiment session. (2) There are many atypical cases (e.g. missing data, strange noise, etc) in the real world which you might not think about. Creating test cases to emulate these real world situations is hard and needs lots of thoughts. (3) Create test cases by generating random numbers (even with some smart restrictions) is not applicable because generated data might not follow real world logic. e.g. brain fMRi signals, heart rate, etc. (4) Even if we randomly generated test cases, we do not know what the correct output should be for that test case. (5) Actually, researchers are writing programs (and functionalities) that never existed before, and nobody knows what the correct output would be.

One possible solution is to learn patterns form existing data, and then synthesis fake data by infusing the learnt patterns. However, this solution doesn't solve the problem very well: how do you know your data generating program is bug-free? Which test cases do you use to test your test-case-generating program?

Another possible solution is to label your test cases (even if it needs lots of human labor), but this solution doesn't solve the whole problem either. Getting large amount of test cases becomes popular in computer vision and machine learning recently. Instead of lacking testing suites, lots of testing datasets (e.g. MNIST, ImageNet) are public available online. Different from traditional software development, machine learning use these testing suites not only as "testing for algorithm" but also as data for training. Nowadays, computer vision and machine learning researchers rely too much on these testing suites, and many research groups are just competing the "testing performance" on those data sets. However, these public testing suites are not sound. For instance, some papers claimed that they can detect vehicles from ImageNet with more than 90% accuracies (also sensitivity and specificity), but their models cannot achieve such performance in real world which contains lots of noises from weather, sunshine/light, shadow/dark, traffic, region/country differences, etc. Hence, even if the program can achieve good result from the testing suites, it does not necessary guarantee the soundness of the algorithm because it cannot be generalized to predict future data.
An obvious possible solution is to "get more data and more variable data", but this solution is not scalable because the real world is so complex. I cannot come up with better solutions for the software development and the machine learning testing problems at this point. However, solving the testing problems becomes crucial in CS research.

Maintenance

Making changes, adding features, resolving bugs for existing code are common tasks in maintenance. Maintenance is unavoidable for both researchers and programmers. At the same time, maintenance is time consuming, dirty, and not intellectually rewarding.

The maintenance problem is also related to "reuse code problem", because good documentation and enough comments can help both "reuse code" and "maintenance". I heard a joke about maintenance, "If you do not comment your code, then nobody can maintain it for you, and you are trapped to maintain your own code for your lifetime until the project is abandoned or comments are made". It's true for some software that the maintenance even takes more time than development.

Maintenance often break previously made assumptions. For instance, I wrote a "Fruit Ninja"-alike mobile game in the past, and now I want to reuse the class Target and class Background, etc in my "AngryBird"-alike video game. However, these class have assumptions that doesn't fit the new game: targets are not round shaped (physical engine change) fruits any more, and the background is not a static image (media codec change) now. If I choose to change existing classes to fit both games, I have to change the class (module) significantly: the logic will become more complex, the code will be longer, it sometimes will break my design pattern, and the UML graph should also be updated. During maintenance, I often found that I hadn't considered lots of things (e.g. multi-language support in game) in the original development. Discussing and thinking deeper about design pattern might help future maintenance, but it cannot avoid maintenance.

Moreover, debugging could cause (generate) more bugs, because debugging process often breaks local logics. Similarly, updating libraries and tools will enforce you to modify your code because of compatibility, which might cause new problems and bugs. Maintain code in local level will also destroy the style of code (e.g. I have to add embedded for loop inside other two for loops). The easiest way to avoid generating new bugs is to use test case, but this method caused testing problem as we discussed above. Read the version change notes before updating libraries is useful to handle library updates, but updating will still causes some unnoticed bugs. Rewrite the whole function might solve the problem of style, but it is time consuming and bug-prone.

In conclusion, in order to maintain code more efficiently and effectively, programmers should think deeper about design before writing, write well-commented code, and read library documentations carefully. New tools should also be developed to help programmers to complete those tasks.