FX Predictor

Machine Learning using Naïve Bayes

Group members

Masakazu Tanami <tanami@g.harvard.edu>

Ben Guild <bguild@fas.harvard.edu>

Ethan Brooks <ethanabrooks@gmail.com>

Muhammad Kaleem Abdullah <kaleem.abdullah@gmail.com>

Original Specification

Final Specification

Versions

Toward the end of the project, the group decided to pursue two separate approaches to the final feature-sets of similar functionality. Accordingly, we have attached two different versions of the project.

Version #1

This version uses modules to handle Data Records and the assignment of categories and attributes. Data derived from training is stored in a hash table for quick insertions and look-ups. Also, this version supports two interpretations of attributes—discrete and continuous—as described in the Final Specification.

Version #2

This version uses more objected-oriented programming with the objective of making the software more extensible and consistent. This enabled the addition of graphing output simultaneous to training data output without duplicating data in the data-structure or manually trying to decipher data from the proprietary Hashtbl key structure used in Version #1. It also introduced the opportunity to subclass or build similar objects based on a parent template for future expandability and inheritance. This branch also experimented with different, perhaps safer methods of obtaining CSV data from files with different CSV formatting using a third-party library and a refined loader for DataRecord.

Individual contributions

We all wore many different hats and were involved with each other's code, but here we can briefly describe some individually-focused efforts.

Ben brought a stronger experience in Git organization and software best-practices. He wrote Main.ml to interface with the Unix environment and provide a lightweight user-experience, while also collaborating with Masa on Version #2's additional features and restructuring of NBModel and DataRecords to bear an object-oriented abstraction and expandability (and native-type passing).

Masa developed the original idea for the project and organized the team. He worked with Ben to develop the first prototype that worked from start to finish. As the teammate with the most experience with financial concepts, he helped the rest of the group understand how to approach the problem of currency prediction.

Ethan wrote the DataRecords module, the Predict module, and the mechanisms used for the continuous interpretation of attributes. He collaborated with Kaleem on version #1's additional features.

Kaleem wrote the original NBModel class used in Version #1 (that also inspired Version #2's) and IO.ml. He helped the team understand the concepts of machine learning and worked on performance optimization.

If we had more time...

We would have built a more sophisticated interface, merged the versions, and spent even more time optimizing in a number of ways. Of course, uploading currency data from a file is not practical if one hopes to predict currency a few minutes in the future. Ideally our interface would pull data off of a website and generated running predictions with a one or two minute outlook. Such advanced user interface mechanisms, however, were far beyond the scope of the class.

Reflections

The original plan, as communicated in the required weekly milestones, served us well, although it inevitably changed over time. The main difficulty that we encountered was that when it came time to implement our favorite additional last-minute features (at the same time as the code was scheduled for cleanup and finalization), we did not adequately coordinate our efforts or communicate our plans as we worked on them. As a result, some new features came to depend on parts of code that had been restructured even with the effort to maintain identical or nearly identical interfaces. Some of said newer elements in simultaneous development still relied on the legacy structure, but were not known to exist in the `master` branch. This is what primarily motivated the decision to split the code into two versions at the very end to demo the additional features separately.

Another difficulty that we encountered along the way was the problem of team members finishing interdependent code at different times. If one teammate finished code that depended on another teammate's unfinished code, an understandable tendency was to move ahead and start working on that section instead of waiting for the second teammate to finish. This often led to conflict subsequently. In reality, different tasks line-up in different ways and flexibility is important, but above all, effective communication is necessary to alleviate these issues.

Our primarily collaboration tool was Git, which a few of us were fairly new to. At first, we each forked the primary repository into separate personal repositories and planned to merge code through merge requests. However, we had difficulty using the merge request interface on SEAS, so we switched to a workflow that used branches in a single repository. One problem was the tendency to accumulate junk files and we advise other groups to come up with some ground rules at the start, especially for the master branch. One rule that we found helpful was that code must compile before merging into the master branch.

In some ways, the team was able to collaborate very effectively on design decisions. There were many instances of cross-talk that led to improvements and we often benefitted from the "another-set-of-eyes" principle. However, there were certain disagreements on design that the team had great difficulty overcoming. Achieving a common design philosophy is difficult in larger groups, especially when communication is hindered by time differences and international boundaries.

With this in mind, one cannot overstress the importance of communication. Our group was spread from Boston to Tokyo and international communication was surprisingly difficult given the time difference and the inability to communicate by phone. Google Hangouts proved to be an inadequate substitute. When forming a group, the ability to communicate should be a *primary* consideration—all the more so in large groups in which coordination takes greater effort.

Advice

First, learn Git early and well. There are many good tutorials online. Remember that accidents on Git can waste serious time and resolving them requires knowledge of the tool, so do not skimp on your education.

Second, in a perfect world, everyone would work on their own modules and code would meet up nicely at interfaces. As discussed previously, we found that this ideal rarely if ever materialized. The solution is communication. Before stepping out of your module, communicate with the teammate who is working on the adjacent module and find an arrangement that works for both of you. And, always pull the latest code from "master" before you start working!

The takeaway

Achieving a common vision of design and a clear division of labor is critical for productivity. If team members do not clarify the design for modules and establish a common understanding (preferably through a reference document, such as a signature or interface) it can create issues down the road for collaboration. Pseudo-code and rough implementations help provide a common touchstone for collaboration and demonstrate concepts that may be hard to discuss without an example. This is only helpful if code is adequately commented and teammates are able to explain their decisions to each other. Finally, before diving straight into major coding efforts, it is crucial to consult with teammates to describe the work you plan to do and the code that it will depend on.