Semester Report

TexIt

Bharath Srivatsan, Rohan Doshi, Andrew Hartnett

May 14th, 2017

Foreword

We were excited but nervous to begin this semester. Building an end-to-end web application for this class wasn't going to be easy, but we were looking forward to getting started nonetheless. The three of us had never been in a group together, but pairs of us had worked on programming assignments and hackathons in the past, and we were excited to see what we could do together over the course of four months.

In the end, the process was challenging but rewarding. We spent many nights watching the sun rise from Robertson basement or Whig hall, and just as many days coding together modifications before our weekly check-ins with Gautam. We had arguments (large and small), but learned how to work as a team to effectively complete tasks. Above all else, though, we were proud to build an application which we hope will genuinely improve the lives of our peers.

Building TexIt has been an honor - we hope you enjoy using it as much as we enjoyed making it.

Bharath, Rohan, and Andrew

The rest of this report proceeds as follows:

1. Planning
2. Stage I: Prototyping
3. Stage II: Core Feature Development
4. Stage III: Deployment
5. Stage IV: Polishing & Testing
6. Frights, Feedback, and the Future
7. Conclusion & Acknowledgements

We apologize for going slightly above the page limit on this report, but wanted to include information we felt was relevant and important to our progress. We've bolded key sections and phrases to help you skim.

Planning

We put a lot of thought into selecting our project idea. To begin, we agreed that **we wanted to build an app that would be broadly usable** - we didn't want our work to die at the end of the course. Second, we wanted a large number of potential users, and didn't want a project that was Princeton-specific. In other words, our vision was to build a project that would, with sufficient time, care, and testing, be launchable on Product Hunt or as a paid service in the 'real' world.

We then got to work brainstorming a huge variety of ideas that would potentially meet these goals. Some highlights included:

- An eight semester course planner (including TigerHub-esque degree progress features), initially personalized for Princeton but eventually generalizable
- A webapp to help high school teachers teach AP Computer Science, including a built-in assignment tool, a code editor and compiler, and an automatic grader
- A photo organization tool that could intelligently allow users to browse memories by linking photos taken in the same place, with the same people, or in the same timeframe

Ultimately, though, we settled on TexIt, an application that would allow students to automatically convert pictures of their problem sets to LaTeX code. For one, **this project seemed to best maximize the tradeoffs between ease-of-construction, usability, and impressiveness**. From our background research, TexIt seemed feasible, but still elicited excitement from our peers when we explained the idea.

Just as importantly, building TexIt would involve writing a large amount of interesting business logic. We didn't want to build an application that simply surfaced stored information; TexIt would give us the challenge of generating unique, high-accuracy image processing and latex conversion methods.

While at the very beginning of the semester we only had rough role outlines, our team eventually settled into a consistent rhythm, with **Andrew on frontend design, Bharath on image processing, latex conversion, and hosting, and Rohan on backend integration and database operations**. These designations were based on our past experience and what we were interested in learning.

Lessons Learned: Break off the project into clean, modular chunks - this applies to both the code and the team! Doing so will give more specialization (and thus, better code), more ownership over individual bugs, and less confusion over responsibilities for tasks. Bonus: fewer messy merge conflicts when all three developers are trying to edit image manipulation code in the same file.

Stage I: Prototyping

We began by deciding on the proposed architecture for our project. At this stage, we didn't diverge far from what we'd outlined in our design document - a frontend written with jQuery and Bootstrap, a Flask app to tie everything together, Python code to preprocess images and call the OCR and math conversion APIs, and a MySQL database to store user information and past files. We did decide, though, to hold off on using React.js or Angular.js to render components; this would have been overkill for our project, as dynamically-rendered flask templates were more than enough for our chosen views.

The big decision to use Flask versus Node.js or Django, etc., was made to maximize compatibility with our pythonic functions, keep the project lightweight, and increase ease of use with various packages (like OpenCV). Generally, **we made design decisions that would simplify our flow and increase effectiveness/accuracy, but that also allowed us to experiment with and learn about useful technologies.** In particular, Bharath wanted to learn about deployment with AWS, Rohan wanted practice with MySQL, and Andrew wanted experience working with material design.

Tips and Tricks: Be very specific about what you want to get out of this course, and structure your project's stack to match those goals! We were uniquely happy with our experience this semester because we came out the other end with specific, demonstrable new skills we'd always wanted to pick up.

Having made those architectural choices, we got to work! We began by building out our core image processing and conversion functionality, thereby meeting our first two milestone goals. This involved us testing four different OCR services, negotiating an API key from and integrating our service with MathPix, and installing and using image processing code written for OpenCV. Of these steps, **we were especially surprised by the weight, complexity, and inflexibility of OpenCV**, an incredibly widely used solution for image manipulation. Unfortunately, our troubles with OpenCV were only just getting started...

Beyond this backend functionality, we designed a basic splash page, and hosted the project on pythonanywhere.com (which came preinstalled with OpenCV!) to meet our third milestone (a simple, ugly user interface).

Lessons Learned: We decided to go 'backend first' by initially focusing on our conversion accuracy before building out our fully-enabled end to end flow. This was probably a bad idea - while it did give us a good amount of time to validate our core technology, we ultimately had to scrap almost all of this code anyways due to deployment bugs. Further, it meant that we didn't have many features to discuss in our first check-ins with Gautam.

Stage II: Core Feature Development

As we moved past week 3, we knew it was crunch time. So far, we'd been building modules separately and locally, focusing on trying a variety of approaches and testing their effectiveness, but we now needed to tie everything together and push it all to the server. Stage III: Deployment, overlapped this stage - we began our deployment process as soon as we had an end-to-end MVP built locally.

Rohan's tasks for these weeks centered around integrating Andrew's views with Bharath's model. By building and sanitizing a whole host of endpoints (including for viewing files, logging in, and uploading images), he **finally made our app work end to end**. On top of this, he set up the databases on Amazon's Relational Database Service (RDS) so that we could finally store and retrieve user information for dynamically rendering views. Having made these changes, we were able to meet our fourth and fifth milestone goals around user logins and stored user content.

There were a large number of UI features we needed to add. Andrew got to work filtering classes from the files page, iterating through four different designs for the page before settling on one that maximized utility while effectively using screenspace. The page looked too empty without a header, but looked awkward without much content to add to the header - **adding class filtering as a drop down was an ideal middle ground**. On the settings page, he worked with Rohan and Bharath to surface basic user information from the database along with an interface to the class creation and deletion endpoints. Finally, he built an early version of what would become the details page, with a rendered image, a convert button, and a code editor for the latex output. By hijacking code from the jCrop library, he was able to build in a box selection flow. At the end of these changes, we considered our sixth milestone (making a prettier UI) complete.

Tips and Tricks: Get user feedback continuously. Early on, we decided to flip our details page to display the image on the left and the latex output on the right because "it seemed more natural" for users to move from left to right.

Bharath spent the majority of his time dealing with deployment issues and porting all of the image processing code to Pillow (see Stage III), but also added a variety of core functions. For one, he dealt with a variety of end-to-end integration tasks, including connecting Andrew's frontend box selection feature with the math conversion methods. He also improved the latex conversion output, embedding the OCR responses in headers and footers to make them latex-compilable. We were still struggling, though, to generate high quality handwriting recognition using Google's OCR service, Cloud Vision.

Stage III: Deployment

After week 2, we knew that we had to port our application from pythonanywhere to a sustainable hosting solution that could dynamically handle large numbers of users. As such, we turned to Amazon's EC2 service, which provides servers-as-a-service for a low cost (in our case, free!). Quickly, Bharath realized that **deploying a flask application was significantly simpler using Elastic Beanstalk**, a different Amazon service that simplified the server initialization and endpoint creation processes.

Lessons Learned: Deployment wasn't even mentioned in our initial milestones. If our experience has taught us anything, it's that deployment can ruin even the best local applications. It's possibly the hardest part of getting an app running. Deploy early!

Disaster quickly ensued. Installing OpenCV involves a long list of complex, fragile dependency installations and compilations, and the only way we'd managed to do this successfully was using a script we'd found that took around an hour and over 2GB of space to complete. Elastic Beanstalk, though, which generated an Amazon AMI instance as opposed to an Ubuntu server, couldn't install many of the core dependencies - nor did it have enough space to complete the installation! Similarly, Google Cloud Vision, our main OCR powerhouse, couldn't install on AWS because of a compilation error in the installation of the required grpcio package.

Almost six weeks in, we were dead in the water. Our image processing and OCR code, the main pieces of our application, simply would not work in the cloud. After spending many hours crawling through forum posts and StackOverflow threads only to see the same errors but no viable solutions, Bharath gave up. **We needed to pivot.**

Luckily, because our code was modular, we were able to quickly start switching out these individual pieces. Bharath began by stripping down the requirements to only what was absolutely necessary, allowing a baseline version to be hosted for the first time. He then replaced the image processing code with new methods that used a different library, Pillow, instead (and then fixed a *new* dependency installation error with that library by manually installing a jpeg encoder). Finally, he wrote a new endpoint that could host images publicly, allowing us to switch the entire text generation flow to use Microsoft's Cloud Vision service instead.

As a happy consequence, our text recognition massively improved - Microsoft's handwriting OCR was miles ahead of Google's; we'd just missed it during our initial trials!

Stage IV: Polishing and Testing

Our push to submission was characterized by a variety of incredibly critical feature changes and bug fixes to polish the final version. We've outlined below some highlights from this week of breakneck development.

Feature Additions/Modifications:

- Save & download flows: We added in critical features to allow users to save their progress after making edits to the latex output, and to download their latex code in a .txt file
- Photo resizing: We now dynamically resize and optimize photos on the backend to increase the image (pixel count) and file (mb) sizes our app can handle
- Optimizing page loads: By eliminating the need for bootstrap in all internal pages, we can render content (like filtered class views) faster
- LaTeX 'special characters': We automatically detect and escape special LaTeX characters like $ or { so that the OCR output is always compilable normally

Bug Fixes:

- LaTeX output that was over half the page height would be automatically hidden by the editor - users who didn't scroll down regularly reported that they thought the code returned was incomplete!
- Images would often show up sideways in our view, even though they'd be the right way up on disk (in Preview). As it turns out, our laptops have built in methods to read image metadata from cellphone cameras and rotate images accordingly - we found and added similar code to our own image rendering system
- Clicking convert without first selecting a box would break the entire details flow; we needed to validate user actions on both the front and back end
- A variety of odd front-end quirks persisted, including menu buttons on the file page that would appear over the upload box, window resizing issues, and and unpredictable challenges with image scaling

Finally, we vastly **improved the security of our app** using a combination of front and back end validation, session cookie checks for AJAX requests, endpoint authorization, etc.

Lessons Learned: Save many minutes for your 'last minute' modifications - there are tons of them you'll uncover.

Frights, Feedback, and the Future

Looking back, there were a few moments that completely took us by surprise. Obviously the difficulties we had with deployment rank on the top of that list - losing access to our main image processing and OCR libraries super late in the semester gave us a massive shock. **We were also surprised to learn, though, how difficult small changes can be to make**, especially on the front end. Tweaks as small as making pages dynamically resizable only down to a certain limit (to prevent content from overlapping itself) ended up taking loads of valuable time and many iterations of potential fixes.

Finally, having never built an OCR-related application before, we were surprised to learn how lacking OCR services are. Even massive providers like Google were unable to return coherent, useful responses for basic inputs; Microsoft's offering was many orders of magnitude better but still struggles with different handwriting styles and other visual inconsistencies. The same issues applied to math recognition from MathPix. While it was heartening to see the API's admirable performance on basic formulae, it quickly broke down on larger inputs and more messy writing (forcing us to limit box sizes and provide more helpful error messages).

Over the final week and a half, but in particular after the demo presentation, we sent out a host of user accounts **to allow family, friends, and strangers to participate in a closed beta**. We learned quite a bit from this process - some things that we've done well (class filtering, file storage, etc.), and things we need to keep working on (text conversion accuracy). We learned that some fixes we'd made, like the loading animation for uploads and the repositioning of the conversion instructions, went a long way towards improving usability. Finally, we even discovered a new potential use case for our application from a friend who had suffered a concussion - allowing users to minimize screen time, while still being able to typeset their assignments and papers.

Tips and Tricks: Create a clear script for your user betas. We used a Google Form with step by step instructions and a host of questions (both multiple-choice and long-form) to generate very specific feedback

Looking forward, we want to deploy this application in the real world. Before that, though, there are a number of features we're working on, like:

- Automatically sending LaTeX code to Overleaf, a web-based LaTeX editor and compiler
- Allowing users to download compiled pdfs as opposed to just .txt latex outputs
- Supporting multi-paged problem sets
- Creating an equation editor box so it's easier to find and paste in converted equations

Conclusion and Acknowledgements

We had a blast building TexIt, even if there were many points along the way at which we brainstormed ways to hunt down the creators of OpenCV, or train our own Google-beating OCR service, or even just chuck our laptops from the CS Tea Room. We'd like to sincerely thank a number of people who stopped us from doing any of those things - our beta users for pretending to be impressed even at our early stage outputs, our TA Gautam for being patient with us and having faith in us through roadblock after roadblock, and Professor Kernighan for sprinkling in just enough helpful hints through the course and on Piazza.

Some final tips!

- Take screenshots as you go along - we wish we had pictures of our initial implementations (mainly to laugh at)
- Stack Overflow is your friend. Bookmark particularly helpful links - you *will* run into the same issues over and over again
- Build something you *want* to build, even if it's very challenging. We were glad we did.