Quickly filled out Nov 1st, expanded on later

Andrew Downie
Post Mortem

Post Mortem

Project Details

Project size

- Using github as a rough estimator, our teams project is ~1400 lines of python
- Fairly small project, but python is a very productive language

Project duration

- Our project took ~3 months
- ~2 months of coding
- Coding took place from very end of September to very end of November (repo migration happened mid October, so GitHub does not show us starting in September)

The software we used includes

- Development stack
  - Python 3.5
  - Psycopg2 - a python library, used to create a connection with a PostGre database
  - All other libraries we used come with python by default (most notably: socket, and json)
- Project management:
  - Git
  - Github
  - Trello
  - Facebook - communication with clinical analysis

Objectives of this project

- To, first and foremost provide data for the Clinical Interface team from the database
- To provide value to clinicians by using the data available to us, to make predictions about the health of a cat, given its characteristics

Successes

- Fully completed the first objective of this project, the Clinical Interface team is able to send us requests for data in a flexible manner, and we are able to get it for them from the database
- Started and provided strong proof of concept for the second objective, we are able to do analysis and predict a few things about cats. Our portion of the project is ready for continually expansion.
- In other words, we created a working prototype, that is flexible to future change
- Recursive json parser, that is able to take apart a request with unlimited layers of logic
- A json setup that allows for a huge amount custom input for what data should be gathered, how it should be manipulated / analyzed, and the form it should be returned in
- Multiplatform, as python and psycopg2 run well on the big three desktop platforms (windows, linux, mac)

- Used python, which we think computing students should prefer over R. Python has the Anaconda package, which brings a lot of the things R can do to python.
- Used sockets, and JSON. Both of which are platform independent, and allow for a great deal of expandability and future flexibility. Sockets allow for the frontend to be entirely decoupled from our part, meaning our part could be swapped out, or the frontend could be swapped out. As long as the json is correct, our part would neither care nor be able to tell the difference. Good support for this is the fact our teams testing of our code happened entirely through sending saved handwritten JSON requests through a socket connection to hit the code we wanted to test. It didn't matter where that JSON came from, it just cares about the JSON.
- Sockets provide another decoupling benefit: since they take place over the network by default, the Clinical interface could be run on one computer, and the Clinical Analysis portion run on another computer / server. OR both the front end and analysis could be run on the local computer, and the front end could simply point to localhost, and the correct port.
- Responsive interaction with clinical interface team, most cases fixing issues within hours, and never taking longer than a couple of days from when the clinical interface team raised an issue

Failures

- Our project, will only handle one request at a time, which is fine when running locally. However if to be used in the cloud, each request should be handed off to a thread, so that one user crashing the app, does not crash the server for all users, and more than one user can be served at a time.
- Would not work with browser based front-ends (could use http or websockets instead of sockets)
- While sockets are very flexible, in our current project state, they are overkill. Sockets allow for cross platform two way communication, between different any combination of supported platform / OS. We don't need the two way part, so we could just use http requests, which would be simpler, and allow our app to connect to browser based front ends (think: if the clinical analysis team did everything in a browser, like service now)
- None of use are trained in analysis, so while we think we did a good job on starting the analysis…. Did we?
- While we think we did a good job setting up this project and coding it… did we? In other words we didn't seek enough outside objective criticism, instead we fed off each other, which leads me to think we're overlooking things an outside set of eyes would see
- While we tried to be easy to work with for the Clinical Frontend team, we could have done better. If I were to do this class over, I would fight to schedule a 20-30 minutes meeting with the Clinical Interface team at least once a week. For multiple reasons,
    - Reason One: would be to make it clear what we are planning to do, so the other team can react, and also understand what to expect. For a while near the start of the semester, I felt like we were holding them up, because they didn't have a clear understanding what they'd be able to do using our part of this project. To be honest though, we didn't know what our part of the app was going to do back then clearly either.
    - Reason Two: so that issues happening when integrating the two teams code has a guaranteed time and place to be discussed in detail every week, and every single member is aware that issues are happening, whether they are directly affected or not.
    - Reason Three: unused time during this weekly 20-30 minute meeting could be used to break into pairs and perform code reviews, where one member of the pair was from one team, and one member of the pair was from the other team. This mixed pair code review would have several benefits:
        - **Improved understanding of the other teams part of the project, improved empathy when things go wrong.** Fortunately nothing failed horribly, and the clinical interface team was very pleasant to deal with. But that doesn't mean such risks should have been ignored.
        - **Improved code, and outside eyes viewing the code.** By having a pair of eyes going over my code every week, I would be guaranteed to take an extra 10 minutes to make it squeaky clean first. All members be on the same team, and working on the same piece, means blind spots may develop. By introducing eyes

that aren't so close to the problems, these outside eyes may be able to see something related to the big picture our group was overlooking, forest for the trees and all that. They might also be able to come up with novel solutions for problems we were trying to solve.

- **Improved inter-team communication.** Inter-team communication consisted of a couple of members from one team talking to a couple members of the other and conveying these conversations to the rest of the group. While this "liaison approach" worked very well for the super group of 31 students, for the two groups that have to have their modules work closely, the liaison approach doesn't seem like a good fit. By merging the two groups into mixed group-pairs for code review once a week, all members of each team would become familiar with at least one member of the other team, improving communication lines. Even without the code review, the weekly meetings would have improved communication of all members between the groups. Since our modules we're not tightly coupled, our method of communication worked well enough in this case, but I would still have prefered more interaction between the teams.

Action Plan

Looking at sockets

Something to keep in mind, is that sockets are currently overkill for our project. Sockets are great when you want low latency, ultra low overhead two way communication. Sockets are expensive to setup, but very cheap to communicate over.

Our project does not use sockets the way they are meant to be used, why?
- We do not need low latency communication (reliable sub 100ms requests)
- We do not need ultra low overhead (We're not sending an integer 30 times a second, we're sending tens to thousands of bytes less than once per second)
- We do not need two way communication
- We create a socket connection for every request, and then throw it away, incurring the most expensive part of a socket, and none of the benefit

Why did we chose sockets then?
- **On the small scale using them sub-properly isn't a big deal**. However we have no idea if our portion of this system would end up on a big server somewhere, serving every user that uses this product. In such a case the overhead of opening and closing a socket for every connection from thousands of users would be massive compared to alternatives, costing way too many resources (cpu and ram) for zero benefit
- **We don't know how request will change, maybe in the future two way communication would be desirable, in which case sockets are perfect.** This would especially be relevant if this product were packaged to entirely run locally. For example, if in the future: data from the database could be cached, and then requests to analyze the cached data could be made in real time, while keeping the frontend and analysis engine decoupled. Running the analysis engine locally would make the overhead of the socket irrelevant. (Hypothetical example: instead of wasting 1mb per socket * 10000 users running this project in the cloud, running it locally mean we're just making the program use an extra megabyte locally)

While we think sockets were a good choice, given the unknowns, there are alternatives that should be considered going forward. I'll briefly discuss a couple I find noteworthy.

Http requests

In the project's current state, a request is made and a result returned. This is textbook definition of client server setup. While sockets are supported pretty much every platform, http requests are absolutely ubiquitous. Changing to http requests would allow the leverage of server tech like Apache or NGINX. Either of these tools would be an excellent choice for turning this project into a durable, scalable server app, with a desktop client. However leveraging these tools means losing the flexibility of easily running the analysis engine locally, and increases overhead when running small numbers of this project.

Http requests could be used without using any server tech (Apache or NGINX), which would give the benefit of using a simple, easy to use, ubiquitous communication technology, while still keeping it simple to run this project both locally and in the cloud. This however would mean loss of the benefits server tech provides and also the loss of the benefits sockets provide (See: “Our project does not use sockets the way they are meant to be used”, at the start of the “Looking at sockets section”).

Web Sockets

Due to security reasons, modern web browsers prohibit the use of standard sockets. Thus websockets were created. Websockets provide the same basic set of benefits that regular sockets do. They are built on top of tcp, just like http is. Being built on tcp means they are safe to use in a browser.

WebSockers work in any environment regular sockets do, but websockets also work in the browser. If the switch to WebSockets were made, we would get the benefits that regular sockets provides, plus the ability to connect to a web browser. Meaning the frontends could be written anywhere they can be now and also browser based frontends could be written. Why the interest in making our project connect to a front end in a web browser? I’ll explain in the next section.

Making this a web app

Something else to consider is how to get users using this application, and what it would look like to support it and maintain it with updates. For this I would suggest following ServiceNow’s approach and making it a web app. (I would also recommend mimicking ServiceNow’s method of composing complex search queries using a GUI, as the user can build the search query dynamically to really flesh out the specific data they want).

With a web app the user does not install anything, and yet gets the full power of a desktop application. The user would simply navigate to a url, and login in. The front end would be loaded dynamically as the user uses the frontend, as the front end **is** the website. The analysis engine would be hosted in the cloud, so the user would still get the full power of a desktop application. This also means the users device is a thin client, so the user would get similar experience regardless of age or power of their device, provided they can install somewhat modern browsers. Updating the frontend could be as simple as the user reloading the page. Updating the backend would be simple as well, as we would have full control of the analysis engine in the cloud.

While cloud hosting would have costs, using something like AWS lambda, would mean only the precise amount of usage would be paid for. Meaning if we had zero analysis performed, costs would be pretty much zero, and only what is used would be paid for. This approach has both the advantage of only paying for what is used, and has the ability to potentially scale to infinity users, bottle necked by the number of users the database could handle. Using something like lamda would make a lot of sense when first going to market. But once users are regular, it would probably make more sense to move to dedicated hosting, so that users pay per time period, not per analysis, because ideally we don’t want our users afraid to use our product, of course the option is there to do pay per use directly to customers. Also dedicated hosting could be acquired for cheap base line capacity, and lamba or other cloud services could be used to fill in missing capacity during surges.

Again while cloud hosting would have a hosting cost, it does save on the deployment and maintenance of the product, and it does make it much easier to deploy the product to as many users as painlessly as possible.

Making the project a web app based product, would make it very easy to get users using it, and to update it. If a free trial were offered to qualified clinicians, then they would just have to sign into the website and they’re good to go. For example we could send emails asking clinicians if they would be interested in product that does the things our product does, and if they said yes, we could automatically provide them with temporary credentials to let them take it for a spin right away.

Regular sockets do not work in a browser, so in order to turn this project into a web app would require using a communication tech that does work in a browser. *Http requests* or *websockets* would both work, and were discussed in an above section.

Analysis

While our project does do predictive analysis, I can't imagine it's anywhere near the level of detail that clinicians would want to see. This is for three main reasons: we are not staticians, we do not have expertise knowledge on cats, and the database is quite limited in terms of the different info about each cat.

We are not staticians
None of the members of our group are staticians, meaning we are not knowledgeable enough to fully leverage the many tools for manipulating data made available by python. Let alone the fact that we did not even touch anaconda.

We do not have expertise knowledge on cats
To form our predictive analysis, we used data from the database itself. The general idea being something like: if we had two cats in the database, and both were grey and one had cancer, our predictive analysis would be along the lines of: if your cat is grey, then it has a 50% chance of cancer. Now this approach would become very powerful if we could properly take into account every trait a cat has, and properly map the cause effect between the traits, and the cat's chances of things like disease. This would require us being statisticians to do it in manner that only needs the data and could be applied to any animal, or it would require us knowing a lot about cat health, so we could build the relevant analysis specifically for cats.

The database is quite limited in terms of the different info provided by each cat
There is a balance to be walked between using info that is available, and asking for information that would be useful about cats, so we could use that info in the future. For now we're stuck with just using the most common info. Although my group members did do some fairly clever things, for example, the way they derived bmi from info we had. To determine volume, they used length, width and height, to construct a cylinder to approximate the head and body. They then divided this volume by weight to estimate bmi, of course this will have wildly fluctuating rates of accuracy, but maybe it's possible to accurately derive a lot more info using the info we already have, in which case the sections below become less relevant.

To really get serious about predicting cat health, I think we would need more information about the cat. Which has two clear problems: one, this info probably doesn't exist at any meaningful size to be useful, and two, it's annoying to get. To solve this, a spin off project could be creating a toolkit that makes it easy to measure every trait about a cat that could be used to predict its health.

As a far out idea, imagine if we used machine learning to allow clinicians to simply take multiple key pictures of a cat, and then the machine learning program would be able to determine color, and breed of the cat. Imagine if this could be extended to the cats: length, width and height. We've gone this far, why stop now: imagine, paw width, leg length, and all other 5000 possible measures you could take of a cat. With this far out idea, you could imagine it would be easy to measure info about strays that are picked up, requiring a few clicks on a camera, and you're done, this ease of use could massively increase accruate and detailed info collection about cats. However when dealing with strays even this far out idea does not address diagnosing sick strays with what diseases they have, as I imagine that would be more costly to perform than taking pictures.

In conclusion, I think this project to go into production would probably need someone good a statistics, someone knowledgeable about cats, and a whole lot more info about cats. As for exactly how to solve the problem of getting more information about cats, I'm not really sure.