Maintenance Plan: Spotify Recommendation Engine

Grant Jurgensen, Stephen Longofono, and Stephen Wiss

This document serves to identify the ongoing process of maintaining our song recommendation software, strategies for doing so, and the associated costs. Beyond the typical software maintenance costs of user assistance, security, and platform changes, we also anticipate updating and adding functionality based on changes to Spotify's API. This includes a contingency plan, in the case that Spotify begins charging for access to it's developer accounts. We also include costs and labor associated with the refinement of our recommendation algorithm, especially as the API endpoints continue to develop.

1. General Maintenance

Speaking in terms of General ongoing Maintenance, our update schedule will be synchronous to that of the Spotify web-API team. That is to say, there wouldn't be a schedule. Judging by the graphed history of commits to the Spotify web-API Github page, it appears that updates and changes to the source code only occur when errors or bugs are discovered. This means that if our team is to be effective at maintaining quality of product and uninterrupted operation for our userbase, then we will need to monitor and be ready to integrate the changes to Spotify's web-API which affect our source code. Such a process could be automated via build tools and scripting, but acting on a failed build would still require a developer. One thing worth noting to our benefit is that our application utilizes Spotipy - an open-source Python module incorporating the Spotify API. This could prove to be either beneficial or costly to us. On one hand, we have a sort of buffer between our source code and any incompatible changes implemented into Spotify's API. So long as the Spotipy developers are prompt in their addressing of issues, and maintain a relatively stable endpoint, many changes might never reach our users. On the other hand, relyingon this strategy means we are dependent on the maintenance and upkeep of an open-source library that appears (based once again on its Github commits) to be dead or dying.

A more practical and realistic plan to combat these inevitable pitfalls would be to employ someone to act as a support representative/maintaining developer. The operative of this position would be two-fold:

- Address known issues. (changes to the API)
- Provide support and feedback in response to user-submitted bugs

This is of course, a reactionary tactic - the provided circumstances do not leave room for a lot of proactivity. It also is a very short list of responsibilities, and would not lend itself well to an efficient use of our funds. However, if the project moves forward according to our Deployment Plan, more work will be required to maintain the additional features added to our application, potentially enough to merit a part-time employee.

It should also be mentioned that our application will need to be updated frequently to remain compatible with current working versions of popular operating systems. Some current features will add further complication to this and as such, We will have to consider switching out some aspects of our application for ones that are more scalable.

2. Potential Maintenance

In it's current state, our application is the opposite of user-friendly. This is fine at present, because our chief demographic are technically-minded users that wish to evolve and fit our source code and algorithm into their own, more specific custom applications. In our deployment plan we mentioned that part of preparing our application for release will include allowing a user access to our algorithm in an effort to stream-line cross-implementation. This was the original intention, and aligns perfectly with the open-source philosophy. However, in order to fund the further development and upkeep of our application it will be necessary to generate more revenue. This is discussed at length in our deployment plan, but in short, with the expected market and sale price of our application, even a part-time employee is too expensive. Maintenance for the beta portion of our release is relatively minor; We would need to have an actively maintained repository on Github with perhaps a "stickied" thread on a technical/programming forum or board (*i.e.* Reddit, Stackexchange, etc...) with general tips and an FAQ.

The other phase of our release, which is the fully functioning song suggester application catering to less technically-conditioned users, will incur a significant amount of cost and maintenance. First of all, The application in it's current state involves running a shell-script to generate the users profile and generate a playlist of songs. As mentioned in our deployment plan, we will have to port the application to IOS and Android in order to make the application saleable. Doing so will alleviate the learning curve necessary to running our application. It will also incur more necessary maintenance. Which will be expanded upon in the next section.

Our Maintenance Plan would also be incomplete without discussing in detail what would happen should Spotify decide to start charging for access to its web API. As is the case with all hypothetical situations, we can't be certain exactly what this would entail, but we can piece together some sort of contingency plan in the event that it happens.

API pricing typically amounts to a tiered pricing, where the differences in price listed are relative to depth of access or computational/spatial complexity of the API access. It's not for certain where our application would fall within this schema, but assuming the worst, this would add a large additional cost to maintaining our application as our user base grows. This would also incentivize us to revisit some portions of our application and make them more efficient in their use of API calls.

3. Maintenance Involving Services, Hosting & Platform

Adapting and integrating our application to a fully-scalable and marketable product will introduce nearly as much work in maintenance as that of the deployment stage. Our application, as is, is written in Python using the Spotipy module. Porting our application over to Android would require rewriting all of our code to Java, and the associated Java web libraries for interacting with the Spotify API endpoint. This would be feasible, taking into consideration the modularity of our algorithm, as well as the extent of libraries available on the Java platform. In the case of making it available to Apple users, the process becomes more involved. The portability of Python to IOS is less accessible to us than for Android, meaning we would need to seek outside help or invest time in learning the platform. Apple has a reputation for a lengthy and stringent approval process for applications that require network access to function, which would introduce potential time and labor costs to the endeavor. If nothin else, we would need to wrap our application in C-Sharp or Swift, and neither of these have included libraries for interfacing a Python web-API. In sum, the initial process of translating our application would require some work (a lot of work in the case of IOS), and maintaining it in it's translated state would require the same breed of maintenance already covered in the General Maintenance section above.

In addition to ensuring compatibility with mobile operating systems, we would need to make sure that our application worked on different desktop operating systems. Fortunately, Python runs natively on Windows, Mac and Linux. We would just need to employ someone to make sure that our application remains up to date with new releases and updates to these operating systems. When combined with the aforementioned needs of API and bug-fix maintenance, the need for a dedicated maintenance developer is better justified.

One way to potentially circumvent porting the code over to cross-platforms would be to host it remotely or simpl turn it into a web-based application. This would require switching over to using something besides SQLite3 for our databases. For example, we could use make use of Amazon Web Service's remote file system. Doing this would introduce more fees for hosting and another engineer to be paid potentially for supporting this side of our application. This would be a lot of work and would be rather expensive but it could potentially save us some hassle down the road and therefore should be considered in the maintenance plan. This consideration is also relevant if our application becomes popular, as a platform-agnostic application can be run on more devices and thus reach more potential users.

4. Projected Costs

In the previous sections, certain maintenance tactics were discussed that will be employed by our team based on what number of the different hypothetical situations come to bear. This section will itemize and compute a rough estimate of the total monthly overhead for maintaining our application.

The backend costs will primarily pertain to salaries for our engineers and QA, as well as hosting fees on whichever platform we choose. To be exhaustive I'll include a few to choose from.

Assuming we hire someone new to manage the QA and the support side of our application (as opposed to doing it ourselves). We will need to pay that person. Due of the lack of technical acuity required, the job itself is relatively non-intensive and could be relegated to a newly-graduated or otherwise entry-level employee. According to the Salary Data & Career Research Center, the average salary in the Kansas City area for a software engineer is approximately $59000 annually. Given that the work would be part time and more akin to a consulting position, we prorate the salary to align with a part-time position. For a conservative estimate that the position will require roughly 10 hours a month, the total cost of employment for our QA department will amount to $280 per month, including taxes and state benefits.

In return for the above compensation, the maintenance team responsibilities would be divided as follows for the period following the full release of our application:

Table 1 Maintenance Position Breakdown

Task	% of time	Monthly Labor
Bugfixes & User Assistance	40	$112.00
Minor API changes	15	$42.00
Security, OS, or platform patches	25	$70.00
Feature Development	20	$56.00

As our user base grows, additional one-off costs associated with scaling our code will be incurred. These can be handled on an as-needed basis, or incrementally by a talented developer in a maintenance position.

5. Conclusion

In closing, The maintenance required for this application will extensive and unpredictable as it is contingent on a number of factors. This exercise has made it plain to see why start-ups always look great in theory, but tend to be a daunting task when it comes down to it. Between the exorbitant fees charged by App retailers in addition to how expensive it is for remote hosting, and the fact that our application will barely draw enough revenue to fund our support staff, let alone turn a profit for the executives (Us). All of this in itself would be insurmountable for a small tech business, but added to the fact that we would need to spend time and resources (and money) to continue growing our application and take care of our userbase means that it in itself is not a feasible revenue generator and would last better as an open-source free-to-use resource for other developers.

References

Pricing-People Data APIPricing-People Data API (2013). "Start for Free. Upgrade When Ready." N.p., n.d. Web. Accessed 14 Dec. 2016.
MasheryMashery (2016). "API Pricing Tiered & Flexible." N.p., n.d. Web. Accessed 14 Dec. 2016.
Salary Data Career Research CenterSalary Data Career Research Center (United States) (2016). " Average Salaries, Job Descriptions, Annual Job Salaries, PayScale." N.p., n.d. Web. Accessed 14 Dec. 2016.