Building a Scalable Data Model with an Interoperable API Architecture for Matching Cancer-Related Clinical Trial Participants

Jacob Strickland
Data Scientist
C2 Labs
Acworth, Georgia (United States)
jstrickland@c2labs.com

Jed Thornock
Senior DevOps Engineer
C2 Labs
Austin, Texas (United States)
jthornock@c2labs.com

Juliette Easley
Data Scientist
C2 Labs
Knoxville, TN (United States)
jeasley@c2labs.com

Travis Howerton
Chief Technology Officer (CTO)
C2 Labs
Knoxville, TN (United States)
thowerton@c2labs.com

***Abstract*—Demonstration of a scale-out architecture for improving the performance and search accuracy of the Cancer.gov clinical trials finder. We leveraged Natural Language Processing (NLP) algorithms to improve the accuracy of search results, big data platforms to improve the performance of data queries, and advanced visualization to improve the usefulness of the data to both clinical trial participants and providers. This architecture is extensible to allow any data scientist to plug their algorithm into the solution to fetch results, perform their calculations, and visualize the results in a common and repeatable pattern. This approach should improve the likelihood of a participant finding the best clinical trial to improve their health outcome while also lowering the cost of providing the service using our scale-out architecture. Finally, it lowers the barrier to entry for performing data science calculations by providing inter-operable Application Programming Interfaces (APIs) where any script/algorithm can be easily inserted into the architecture without worrying about how it will access or visualize the data.**

***Index Terms*—Scale-out, NLP, cancer, clinical trials, big data, APIs**

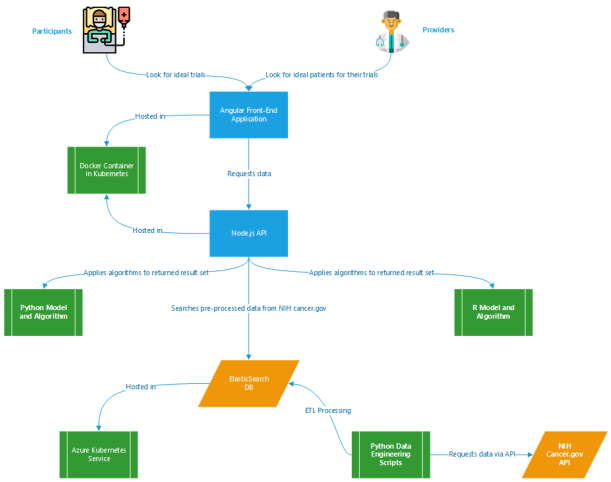

Fig. 1. Solution Architecture

I. Introduction

Cancer is a life-threatening illness affecting over 1.7 million people in the United States (US) on an annual basis and resulting in over 609,000 deaths. [3] As the cancer progresses through the various stages, the medical prognosis becomes increasingly dire and affected patients turn to sites such as Cancer.gov to locate clinical trials that may perform life saving interventions. Since clinical trials are often the last hope for some cancer patients, it is imperative that the US provide access to the most timely and relevant clinical trials that offer the best chance of a positive intervention. Therefore, our team developed a proof of concept solution architecture as shown in Figure 1.

In developing this architecture, we established the following goals that would govern a successful solution:

- Performance at Scale - the solution must be able to scale to support Petabyte sized data sets without the need to re-architect the solution or leverage specialized hardware
- Horizontal Scale - each component of the architecture should be able to scale horizontally to handle load; independent of any other layer of the architecture
- Cost Optimization - each component of the solution should leverage free and preferably Open Source technologies to allow the solution to be scaled cost-effectively without concerns around licensing
- Interoperability - the API layer should easily integrate any

data science related script, wrap it with an addressable API endpoint, and provide access to both back-end data and interactive front-end visualization; scaling "to the n" to support an unlimited number of potential algorithms and APIs
- User Experience - the solution should provide a rich set of interactive visualizations and a common way to view results across multiple algorithm/API calculations
- Relevance - the algorithms leveraged should improve the accuracy of clinical trial results for any given participant over the existing results available on the Cancer.gov website

This paper will describe how C2 Labs has achieved these results by ingesting large clinical trial data sets leveraging commodity cloud hardware (light-weight Virtual Machines (VMs)) and open source software, with two example algorithms provided to improve the accuracy of the results (and providing a basic proof point for multi-API inter-operability), and a front-end application to visualize the results.

II. Data

A. Original Data

The original data was synthetic data provided by the Smokey Mountain Data Challenge (Challenge 6) for 100 cancer patients [4], detailed eligibility criteria for many different trials, and patient scores on each trial for 10 patients. This data was stored in three zip files. The synthetic patient data included things like age, gender, cancer location, Hemoglobin level (Hg), white blood cell count (WBC), platelet count, and other details about the participants' specific medical attributes. The eligibility criteria included data for trials with six files detailing eligibility based off various criteria. The patients received a global metric that was given based off their eligibility for a trial and this data was included in ten Microsoft Excel files, and each file consisted of a single participant's scores for each trial.

B. National Cancer Institute (NCI) Data

Our team found that the data provided was insufficient to meet the overall goals set for this challenge. Additional NCI data was ingested via APIs to enrich the data and gain more information about the trials from the original data set. Due to resource constraints within our Kubernetes cluster (and the slowness of the Cancer.gov APIs), we limited our list of trials to the original data set, looped over each trial, and fed the trial number into the NCI API to extract all of the data related to each trial. This data was then ingested in our ElasticSearch instance, various indexes were built, visualizations were created, and data cleansing was performed to result in a workable data set for the prototype delivered in this challenge.

III. Tools

A. Elasticsearch

The back-end of the solution is based on ElasticSearch which is a big data platform for storing, search, and visualizing data in a secure and highly performant manner. Our ElasticSearch solution was hosted on top of Kubernetes to allow for ease of scalability as the performance and storage needs increase over time. The advantages of this platform include:

- Free and open source solution for interacting with data at Petabyte+ scale
- High performance indexing for searching large data sets
- Large ecosystem of add-on products that can integrate with the platform over time
- Support for Representational State Transfer (REST) API endpoints providing for standards based interoperability with other systems

This solution was hosted in our existing Kubernetes cluster. This cluster is a shared service within our company hosting multiple applications and services (54 existing pods) where this solution scheduled a small amount of excess capacity available within our three node cluster (total capacity of 12 CPUs with 48 GB of RAM). The total cost of hosting this Kubernetes cluster within Azure is less than $200/month, with this solution using less than 25% of that capacity (or less than $50/month), making it extremely cost effective for scale out.

B. Kibana: Data Visualization and Discovery

Kibana is an advanced set of data visualization tools that integrate natively with ElasticSearch. They are frequently installed together in what is commonly referred to as an ELK stack (ElasticSearch, Logstash, and Kibana). By combining a scalable back end data storage and indexing solution with a robust data visualization solution, we were able to effectively meet all of our backend and visualization requirements in a single tool.

As a proof of concept, we conducted analysis to build sample Geographic Information System (GIS) maps that gave a visualization of the clinical trials' geographic spread. Location is an important factor in deciding on a clinical trial, as most are ongoing, and it is generally impractical to recommend a clinical option that is several hundred miles away from a patient's home. Similarly, those looking for participants in a trial must search for patients within a reasonable distance.

Heat Maps: Heat maps were effective in creating visualizations of areas based on the number of trials taking place. See Figure 2.

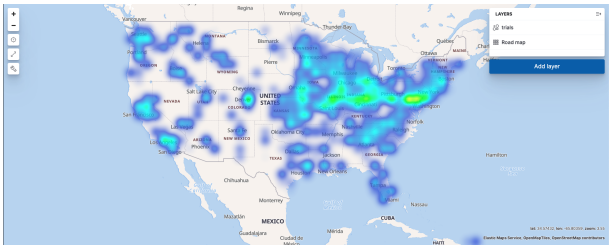

Fig. 2. Trial Heat Map

Cluster Maps: Using longitudinal/latitudinal coordinates of each trial, cluster maps were advantageous in discovering exactly where specific trials are taking place. Ease of zooming

and re-scaling came with the user-friendly attributes of Kibana. See Figure 3.

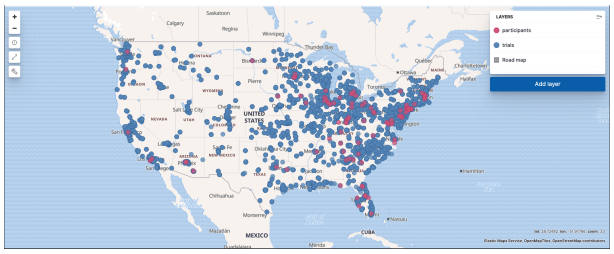

Fig. 3. Trial and Participant Map

C. Docker: Containerization

Portability was a key design consideration in conducting the Proof of Concept (POC). Since this solution was built using company infrastructure and would need to be scaled later, we architected a containerized solution that bundles all application dependencies into Docker containers that can be easily ported to any infrastructure hosting solution; whether on-premise or cloud. The advantages of containerization include:

- Free and open source solution for containerizing applications and micro-services
- Allows for horizontal scalability using container orchestrators (i.e. Kubernetes)
- Allows the solution to be easily de-composed into microservices; each of which can be horizontally scaled
- Containers can be easily ported across infrastructures by minimizing dependencies on the underlying Operating System (OS)

D. Kubernetes: Scale Out

Once the application and related APIs are hosted in containers, there is a need to manage the complexity of interactions between the containers over time as part of operations and maintenance. For our orchestration solution, we implemented Kubernetes, an open source solution from Google. The advantages of Kubernetes included:

- Free and open source solution that is actively maintained by Google
- Provides auto-scaling functionality to increase or decrease the number of pods running based on load metrics
- Provides security features for advanced segmentation using their Software Defined Network (SDN)
- Provides capabilities for non-disruptive upgrades and A/B testing to facilitate rapid scientific discovery through iterative improvements

All tools supporting this project are securely and cost-effectively hosted and scaled within the Kubernetes cluster as shown in Figure 4.

E. Microsoft Azure: Cloud Hosting

Behind the Kubernetes orchestrator, we required a scalable cloud infrastructure for hosting the Kubernetes service. For this requirement, we leveraged Microsoft Azure; primarily

Fig. 4. Kubernetes Cluster

based on having significant in-house expertise in building enterprise-grade solutions on the Microsoft cloud. However, Kubernetes is agnostic to the underlying cloud infrastructure and could just as easily be hosted in Amazon Web Services, Google Cloud, or an on-premise data center. We primarily chose the Azure cloud due to:

- The open source nature of the project and the need to make the data and results publicly available
- Easier to spin up and down infrastructure resources on demand to optimize costs during the POC
- Unlike most cloud services, Azure does not charge for running the Kubernetes service. They only charge for the supporting infrastructure providing significant cost savings for this POC
- We had an available "reserved instance" cluster where we could use existing excess capacity for this experiment without adding cost

F. GitHub: Source Code Management and Open Source Participation

Another requirement for our POC was to iterate on the solution collaboratively in an Agile manner. To do so, we chose GitHub to provide a Git-based workflow for real-time collaboration and Source Code Management (SCM). This solution provides SCM as well as a free platform for collaborating on open source projects. GitHub was chosen due to:

- The open source nature of the project and our desire to collaborate with others in the future to extend the capabilities of the solution
- It is the current standard SCM solution used within our company
- Ability to leverage Git to allow multiple concurrent development activities for rapid iteration

The repository for this project is shown in Figure 5 and the source code is publicly accessible at the following link.

G. Azure DevOps: Continuous Integration and Deployment (CI/CD)

Another key design goal was to eliminate the need to manually build and deploy containers into Kubernetes to test new code. To provide enhanced automation, we built an automated CI/CD pipeline in Microsoft Azure DevOps. This tool allows us to:

- Automate builds and deployments of the Angular application and related APIs
- Leverage our standard build scripts and expertise using the company standard CI/CD solution

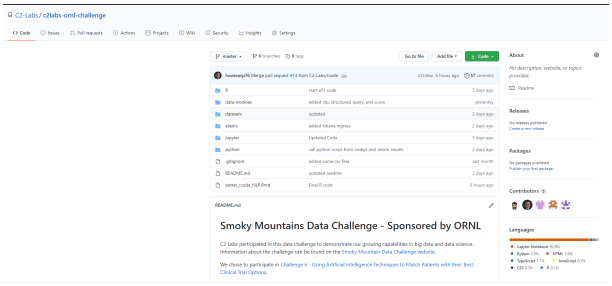

Fig. 5. GitHub Repository

- The solution provides tight integration with Azure, minimizing custom code, to allow the pipeline to be quickly built
- Supports rapid iteration by data scientists on algorithms while eliminating the need to worry about build and deployment tasks

H. Node.j and Express.js: APIs

Interoperability via REST APIs was a core component of our solution architecture. For this purpose, we selected Express.js as our middle-tier server which is based on Node.js (the most popular Javascript framework). There were multiple reasons for choosing Express.js and Node.js which are discussed below:

- Lightweight and easily deployed within Docker containers
- Ability to interact with the data and algorithms via command line and API URL versus just the Graphical User Interface (GUI) in Angular
- Ability to spawn child processes to run Python and R scripts
- Speed in which APIs can be rapidly created
- Integration with Swagger for robust API documentation and testing

I. Angular: Front End Visualization and End User Interaction

The final tool provided a front end application to gather user input and to visualize the results from the data models and APIs. We chose Angular as the front-end Single Page Application (SPA) due to the following reasons:

- Angular is a highly opinionated framework meaning that it ships with many libraries that force developers to do things the "Angular way". This approach provides strong consistency in code between developers which is important when building an extensible architecture with many potential future developers. Angular should lower the cost of maintaining the code over time with a large and distributed developer base for future algorithms.
- Strong capabilities for performing asynchronous API queries while providing strongly typed objects via Typescript
- Compiles to static code allowing the application to be hosted within containers or any object store (i.e. AWS S3 buckets) in a serverless architecture

The Angular application provides the most tangible end user experience and allows potential participants to find the ideal clinical trial. The application starts with a landing page that shows a list of options along with the solution architecture for how the solution was constructed, see Figure 6.

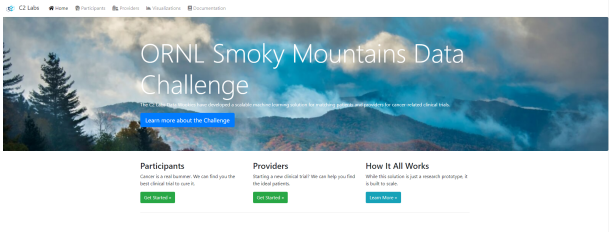

Fig. 6. Landing Page

A potential clinical trial participant can then click the "Get Started" button to bring up the Participant Form, see Figure 7. On this form, they will provide basic metadata such as anatomical site, gender, zipcode, distance they are willing to travel, age, cancer type (can be multiple), and the cancer stage. This metadata is used by our algorithms to optimize the search results to identify the best possible clinical trial options.

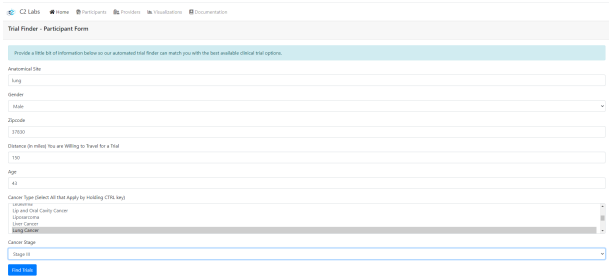

Fig. 7. Participant Form

Once the user clicks "Find Trials", the Angular application kicks off asynchronous API calls that do the following:

- Fetch the potential list of most relevant trials from ElasticSearch based on the criteria provided
- Returns the result set to the API where it then spawns child processes to provide intelligent scoring on the best potential match using both the Python and R algorithms developed and discussed in the next section
- Final results are then returned to the Angular application for visualization

The Angular application then displays the relevant list of search results as shown in Figures 8 and 9.

The User Interface then provides visual indicators on the likelihood of a positive match using the following logic:

- Green trials have a score >= 1.5 and indicate a high likelihood the trial is a good fit for the participant
- Yellow trials have a score < 1.5 and >= 1.0 and indicate a moderate likelihood the trial is a good fit for the participant
- Red trials have a score < 1 and indicate a low likelihood the trial is a good fit for the participant

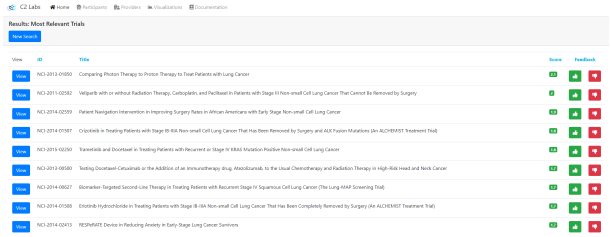

Fig. 8. Trial Search Results - Top Results

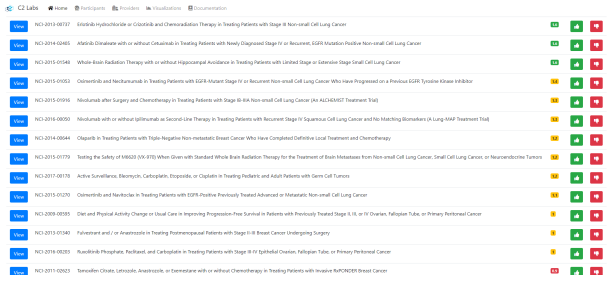

Fig. 9. Trial Search Results - Low Results

The end result is an intuitive and optimized system to provide the best possible clinical trial options. In addition, the UI provides the following useful features:

- "View" button to direct the user to additional information about the trial on the Cancer.gov site
- Thumbs Up/Down buttons to provide feedback on whether the results were useful. These buttons provide a feedback mechanism to the algorithms for future machine learning efforts.

IV. Algorithms

We focused on developing two different algorithms for performing data science work related to improving search results to match participants to clinical trials. This POC was intended to demonstrate that the architecture can support multiple algorithms that are executed in parallel by the middle tier (API layer) and returned to the front end where they are combined for visualization. This architecture could be scaled "to the n" to support many different algorithms in the future that can be wrapped in APIs for consumption by the front end Angular application. This approach would allow researchers from across the globe to collaborate on improving their algorithms without having to worry about implementation details around data storage, retrieval, and visualization. The goal was to lower the barrier to entry for data scientists to work with these large data sets; thus improving the likelihood that progress on optimization algorithms can be made more quickly.

The approach taken for our algorithms and the initial results are discussed in the following section. We intentionally conducted research using two different technologies (Python and R respectively) to demonstrate that the solution is technology agnostic and fully extensible.

A. Python Algorithm

The algorithm that we implemented was inspired by other approximate string matching algorithms. First, we began by extracting some basic unstructured text data which we assumed would be useful in analysis. We decided to extract the descriptions of the trials' eligibility criteria and the cancer's location. We took the extracted data and loaded each piece into two dimensional (2D) arrays. This array contained anatomic sites and descriptions that allowed fuzzy matching to correlate keyword strings like cancer type or stage to descriptions or strings of the unstructured eligibility criteria.

Fuzzy matching has a few different useful applications. It is often used in machine assisted translation of text data. For example, suppose we have a data set that contains records of many company names. The entry in the database should look like the string "Apple Inc." but users may enter the string "APL INC" or "Apple" when searching. We know the two strings mean the same thing, but a computer or searching algorithm would generally treat them differently. This particular problem is where fuzzy matching becomes useful. It takes and compares strings that are not a 100 percent match and finds a correspondence between them. The strings can be compared at the sentence, phrase, and/or word level.

We used methods from the fuzzywuzzy package in python. [5] The fuzz.token_set_ratio function is used to compare the patient data to the trial data. This function takes two strings as arguments. The function then finds all the alphanumeric tokens in the string and treats them as a set of the form (sorted intersection of the strings, then sorted remaining strings). The score between the two strings is then calculated. The library provides controls for comparing specific strings like "brain" and "he has a big brain". [5] The score will be statistically meaningful for the two strings because they both have the term "brain".

We decided to pre-process the descriptions field by removing punctuation and stop words. We know we lose information when we remove stop words, but we can easily fix that by creating our own list of stop words. By doing that, we removed the words that were unnecessary while retaining the relevant information. We used a scoring method based on the score given by the Levenshtein distance or the similarity ratio. [6] For the descriptions field, we incremented the description metric by 0.1 if the score calculated by the function fuzz.token_set_ratio is greater than a specific value. Usually the value should range from 65-75 to get the best results. We used this scoring method since we assumed that different descriptions can have identical tokens, and those repeats increase the score each time they are matched. It does not happen often but that is why we weighed the metric for the descriptions data lower than what we set for the anatomic sites. The anatomic sites had a weight of 1 for every match. We assumed that giving the anatomic site a higher score increased our chance for a better match. After we generated the scores for both the descriptions and the anatomic sites, we added the two scores together to get the final score for the trial. We

loaded the final scores and the trial NCI-IDs into their own lists. We then used a dictionary comprehension in order to store the trials and their scores together. Lastly, we sort the trials by their score and print the final result.

While the algorithm utilized the Levenshtein similarity ratio and leveraged the popular fuzzywuzzy package in Python, we added a unique scoring system that was optimized for this business problem to rank trials for cancer patient data.

B. R Algorithm

For our R algorithm, we tested methods like tokenization [7] and comparing Term Frequency-Inverse Document Frequency (TFIDFs) from the tidytext package [8], and pairwise similarity from the widyr package [9]. After testing through the accuracy of each algorithm, we ultimately chose a scoring method similar to that of the Python code. This code only uses two R packages: stringr [10] and readr [11], and has less than 50 total lines of code.

First, the R code takes the user inputs of cancer stage, anatomic site, and a few keywords. The R code cleans up this input data. It converts integer strings into roman numeral strings, respective of the cancer stage. It then creates a vector whose elements are the keywords, and creates a new element when it sees a comma. Next, the algorithm receives a file of filtered data. The data is filtered and flattened beforehand to only give trials that are relevant to the user's age, location, and gender. The file that the R code receives consists of columns filled with the trial's NCI ID, anatomic site, and several description columns.

Next, we create a column of scores corresponding to each trial, all initially set to zero. Then, each row (trial) is input into a for loop. The loop starts with a statement that gives the trial a score of 1 only if the anatomic site matches. (Note: a trial will continue through the rest of the loop, and receive a final score greater than zero, only if the anatomic site matches.) The rest of the loop iterates through the keywords and stage matches. If the cancer stage matches, .5 is added to the score. Each time a keyword string matches in a description, .1 is added to the score. Finally, the trials are arranged from highest to lowest score, and the user can view trials most relevant to their input.

This code is efficient due to its simplicity and straightforwardness in string matching. It can handle large volumes of data and outputs results quickly. With the help of a dictionary for more complex string matching (i.e., "ovary" vs. "ovarian"), the accuracy of this algorithm can improve further.

V. Architecture

The key to our solution is the loosely-coupled and horizontally scalable architecture. Each component of the architecture is described below:

The key pieces include:

- Angular application - can horizontally scale the number of pods to handle changes in load
- Node.js - each API can be deployed as a micro-service that can be independently scaled to an unlimited number of APIs; each of which can be independently scaled based on load and performance requirements
- ElasticSearch - hosted within Kubernetes allowing additional pods and storage to be added to handle nearly unlimited data sizes
- Microsoft Azure - leverages commodity compute within the Azure cloud to dynamically scale up and down compute, networking, and storage infrastructure based on load

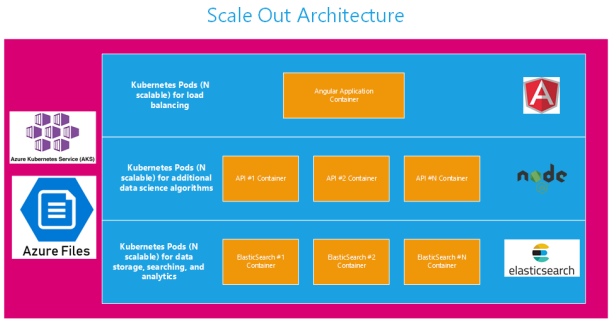

Fig. 10. Scale-Out Architecture

The result of this architecture is an infinitely extensible solution that is constrained only by costs and budget. All of the components are loosely coupled based on REST APIs to easily interchange data. Because of this loose coupling, each layer can scale independently based on changing resource demands without impacting the other layers. In addition, all layers are deployed within Docker containers making them easily portable to on-premise environments or other cloud providers such as Amazon Web Service (AWS) or Google Cloud based on organizational preference.

Conclusion

Our team was able to demonstrate an end to end prototype as a Minimally Viable Product (MVP) that achieved all of the stated goals. The solution is scalable, cost effective, more accurate than the existing Cancer.gov results, intuitive to use, extensible for use by other data scientists for future algorithm development, and freely available as an open source project on the C2 Labs Data Wookies GitHub Repository. In addition, an interactive prototype is available at the Data Wookies website. Some our key results included:

- Significant improvements in accuracy using our Python algorithm versus the Cancer.gov baseline
- Significant improvements in accuracy using our R algorithm versus the Cancer.gov baseline
- Significant improvements in API performance using ElasticSearch and Node.js/Express.js versus the Cancer.gov baseline APIs
- Demonstration of a multi-API integration in the front end visualization with 2 example APIs/algorithms
- Demonstration of GIS-based visualizations and heat maps for large clinical trial data sets

- Delivery of an interactive Angular prototype with optimized results leveraging a limited trial data set (less than 300 trials)

While the initial results are promising, there is much work that could be done to build out this architecture over time to demonstrate even greater results. Examples of future work could include:

- Kubernetes Scaling - our MVP was limited based on the available capacity in our existing Azure Kubernetes cluster. By adding additional VMs to the cluster, the prototype could be scaled with additional pods to handle the full Cancer.gov data set versus the limited set used in the prototype. In addition, this POC did a one time ingest and future efforts would provide automated scripts to keep the data set evergreen over time.
- Additional Algorithms - this architecture allows any future algorithm to be easily inserted as an API to interact with the data set and return results to the front end Angular application. Each new API/algorithm could be hosted in a separate container and horizontally scaled in Kubernetes to allow unlimited numbers of data scientists to develop and quickly deploy new algorithms. For instance, in the future we could create a Bag of Words (BOW) or a Term Frequency Inverse Document Frequency (TF-IDF) model in order to group similar trials with one another for more and/or better recommendations.
- Inversing - our current algorithms use NLP to effectively match participants to the best available clinical trials. We would also propose applying algorithms to perform the inverse calculations and match the ideal participants to clinical trial providers.
- Machine Learning - the Angular application provides the capability for end users to provide feedback using intuitive "thumbs up" or "thumbs down" on each result returned. By providing user based data classification on the quality of results, this approach would allow future development to apply machine learning to analyze which characteristics are most closely correlated to positive results; and conversely, which results are most closely correlated with bad results. This machine learning could be applied to the scoring algorithms to provide improved accuracy over time based on an integrated feedback loop for continuous improvement.
- Automated Testing - the solution does not currently contain any unit tests or end to end testing that can be automated. Future efforts would improve resiliency and quality by building a fully automated test suite that could be executed as part of the CI/CD pipeline.
- High Performance Computing (HPC) - when integrating a larger data set and leveraging more computationally intensive machine learning algorithms, HPC resources could be leveraged to improve both the accuracy of the algorithms and to find novel approaches that would not be computationally possible or cost effective using commodity cloud infrastructure.

To learn more about our solution:

- View our source code and additional documentation on GitHub
- Test it yourself in our interactive web application

CONTRIBUTIONS

Publishing this paper and delivering a functional prototype was a team effort. The specific contributions of each team member are shown below:

- Travis Howerton - Angular application development, lead solution architect, and primary report author
- Jed Thornock - data cleansing and visualization, deployment of the ELK stack on Kubernetes, development of the middle tier APIs, and report contributor
- Jacob Strickland - development of the Python algorithm and report contributor
- Juliette Easley - development of the R algorithm and report contributor

REFERENCES

[1] G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.
[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.
[3] Cancer.gov, Cancer Statistics
[4] Smokey Mountain Data Challenge Website, Challenge 6
[5] Python Package, fuzzywuzzy 0.18.0
[6] Wikipedia, Levenshtein distance
[7] Lincoln Mullen, Introduction to the tokenizers Package
[8] R Tidy Text Package, tidytext
[9] R Widyr Package, widyr
[10] R Stringr Package, stringr
[11] R Readr Package, readr