Paper ET13

Health Equity Explorer: Technical Deep Dive into an R Shiny Application for Revealing Structural Inequity and Systematic Racism in Healthcare

Pavel Demin, Appsilon, Braga, Portugal
William G. Adams, Boston Medical Center, Boston, USA

ABSTRACT

Health Equity Explorer (H2E) is an R/Shiny application that presents various statistical and exploratory tools to analyze aggregated patient records combined with place-based data. Users are able to run regressions, construct time series charts, and interactive geospatial visualizations such as bi-scale and dual comparison maps. This application is designed to be used in different medical institutions, with their own infrastructure and data as long as the data architecture adheres to the OMOP Common Data Model. To maximize the ease of adoption, H2E is built as a dockerized solution that can be deployed on Linux or Windows with a single script with the only dependencies being Docker engine and Bash or Powershell, respectively. The bootstrapping process creates all required infrastructure: Shinyproxy to serve the application, telemetry and validation dashboards, utility database to store logs, Redis for caching, and the development database, if necessary - which is populated from a synthetic backup. When deployed in production, Health Equity Explorer employs a flexible authentication system powered by the capabilities of Shinyproxy.

INTRODUCTION

Health Equity Explorer is an open-source R/Shiny application written with the *Rhino* framework designed to enable users to explore health equity data in an interactive way by building graphs, tables, and maps and conducting statistical analyses [1]. This paper describes the outcome and technical highlights of the second iteration of Health Equity Explorer development in which Appsilon took part. During this time the developers focused on improving the deployment process, overall performance, as well as adding new features like interactive maps, support for hundreds of health equity dimensions, and the ability to create customized extensions to the application.

First, the system design aspect of Health Equity Explorer is described. Usually, R/Shiny applications are designed to be deployed on one centralized server, over which the developers usually have some extent of control. However, this project is designed to be used by different medical and research institutions, making it a “distributable” software. The only reliable way to distribute non-compilable software is to dockerize it. This way the developers are able to minimize the prerequisites and requirements for deployment, while also maximizing the control over the environment in which the application is being executed.

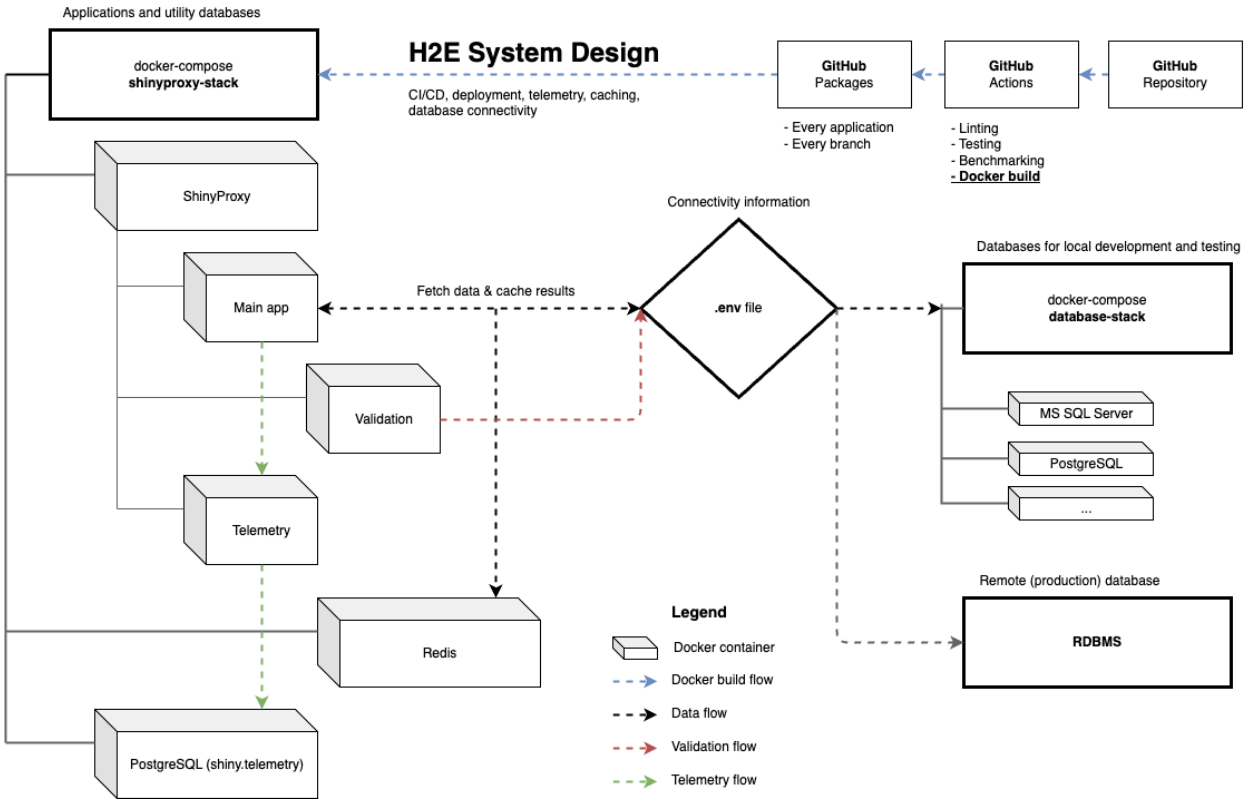
Then, some of the technical highlights of the Shiny application are discussed, starting with the performance optimization process with the details of specific techniques used to make the application faster. Afterwards, the paper demonstrates how to solve a non-trivial user requirement by implementing a high performance React component and making it available in Shiny. Finally, the ability to extend H2E via the Machine Learning Framework is presented.

SYSTEM DESIGN OVERVIEW

Containerization is a critical pre-requisites for deployment of complex applications. Especially, when the deployment goes beyond the application itself: other supporting applications, utility databases, caching database, development database, etc. In certain cases it could also be an authentication service, a computation service, and so on. What’s more, containerized deployments provide greater control over the environment where the application is going to run. Local development, Linux deployment, Windows deployment, usage of different databases - all of this is possible thanks to the complete dockerization of the process. We use Docker because it is one of the most popular solutions for containerization. For development purposes we rely on Docker Desktop: it allows easy inspection of resource usage, logs and container filesystems, debugging of applications at runtime, and execution of arbitrary code inside the container. Apart from Docker we also use docker-compose to launch multiple services at once, create reusable networks and volumes, control environment variables, and more.

Docker images and containers are at the core of this project - images are built in the cloud and then pulled onto the deployment server, where Shinyproxy - running in a container - can discover these images to start new application containers. The diagram below demonstrates the complete architecture of the Health Equity Explorer (H2E). Most of it - excluding Github and remote RDBMS - is bootstrapped with a single bash/powershell script that can be used both for scaffolding development environments and starting a production deployment. Different parts of this diagram will be discussed in the following paragraphs.

Fig. 1. Architecture diagram



SOURCE CODE AND BUILD PROCESS

We begin the overview of Health Equity Explorer architecture from the cloud. First, developers push their code to the Github Repository. For every open pull request Github Actions service runs linters, tests, benchmarks, as well as builds the docker images for all three applications - H2E itself, telemetry dashboard, validation dashboard. These images are then pushed to the Github Container Registry (GHCR), also known as Github Packages. Building images in the cloud is optional, but very much desirable. Depending on the system, it might take up to an hour to build a complex Shiny application from scratch. But pulling an image from the cloud is much faster. Furthermore, the Github Action developed by the Docker team allows developers to leverage Github Artifacts to cache docker builds. Thus, a rebuild of the application only takes a few minutes.

SHINYPROXY

One of the prerequisite requirements for this project has always been the reliance on free and open-source software. Shinyproxy is the deployment of choice for Health Equity Explorer because it is free, comes with authentication and session management. It runs as a service in a docker-compose stack using the main docker network. Thus every new container will have access to the database, redis container, and the telemetry database to write usage logs.

It is not reflected directly on the diagram, but authentication is also an important part of the deployment. Shinyproxy allows us to build a very flexible user management system. Due to the fact that the production application deployed on premises of a medical institution contains very sensitive data, access to it should be limited. By default we achieve this by reading user group information that comes from Active Directory and pass it to the *access-groups* field of Shinyproxy container specification. However, we also take advantage of the *access-expression* field that allows us to check if the user ID from Active Directory is part of a custom user list that we defined. This is useful in two ways: either to overcome bureaucratic blockers, e.g. give person access while the group provisioning is in progress, or the opposite - to limit access to the application only to certain users within a given access group.

Another Shinyproxy feature that we use to improve compliance when working with sensitive data might seem unexpected. It is the ability to provide custom HTML templates for the login page. We added a mandatory “I agree…” checkbox to the login form with a link to the internal rules of responsible data usage. This simple action guarantees that whoever accesses the app must have read the rules of responsible data usage. Thus adding one more security layer, even if a user is defined in the Active Directory and belongs to a group that has access to the application.

DATA MANAGEMENT

Health Equity Explorer is built around the data that is, metaphorically speaking, “blessed” and “cursed” at the same time. On one hand, patient records data that is being analyzed and visualized in the application is so sensitive that only certain individuals in certain organizations can have access to it. Exposure and distribution of such data is legally restricted. Such limitations often mean that the development of applications with such data can only be done

in-house. However, the database hosting this data is created in accordance with The Observational Medical Outcomes Partnership (OMOP) Common Data Model (CDM) - an open community data standard, designed to standardize the structure and content of observational data and to enable efficient analyses that can produce reliable evidence [2]. It means two things: data can be easily synthesized and reproduced. For the Health Equity Explorer as a project it also means that it can be deployed with little-to-no friction wherever this data model is adopted.

On the architecture diagram one can see the two sources of the data: a docker-compose stack with different databases, and the remote Database Management System. The software development team has been working on the project using synthetic backups of the real data, thus being able to reliably reproduce real world usage of the application. This approach also reduces the necessity to create artificial interfaces and mocks for communication with the database. Instead, the local development version of the application is able to “talk” to a very real database. And once it’s time to deploy Health Equity Explorer on the premises, database connectivity credentials are configured in the *.env* file - special file with secrets and variables, akin to *.Renviron* - to point the application to an existing database and make sure that a fresh one is not being created during the bootstrap process.

What is more, local development allows testing the application against different database engines. Docker-compose allows to specify different profiles, based on which a docker container with the database is started, and then populated from an appropriate backup file. While MS SQL Server remains a popular RDBMS solution, we wanted to provide support for at least one popular open source database - PostgreSQL. From the application perspective support for different SQL flavors is provided by the *dbplyr* package. However, not all operations and queries work the same in Postgres and SQL Server, so continuous automated testing is required to ensure correct compatibility.

DATA VALIDATION

Despite all the efforts towards data standardization and conformity to the common data model, development of an application exclusively with mock/synthetic data is still a very high risk. In regulated environments where there is no access to the real data due to its sensitive nature, the best we can do is try to verify the data-related assumptions at runtime. However, instrumenting the application with numerous assertions is not only cumbersome, but also not future proof. If data requirements change, somebody has to crawl through the entire code base to update every imaginary *assert_that()* expression.

Instead, we employ a popular software development principle: “it is easier to ask for forgiveness than permission”. In other words, the application assumes that the data is exactly what it should be and the verification of the assumptions is offloaded to a specialized dashboard. Using Shiny and the *pointblank* package we developed a standalone application for data validation. It has a set of rules such as: existence of certain tables, presence of certain columns in those tables, their data types, minimum number of unique values, etc. This application is deployed within the same Shinyproxy instance and connects to the same database as the main application. When deploying the application for the first time on the new premises, the administrator is able to open the validation app, perform all the checks and identify critical data problems that have the potential to crash the Health Equity Explorer.

CACHING

Caching reactives and outputs is a very powerful technique to optimize Shiny applications. Shiny provides several options for caching, including memory, disk and custom cache implementation - there even is an example implementation of Redis cache [3]. In-memory caching strategy loses most of its value when used with Shinyproxy, so really the options that we have are disk cache or Redis. Retrospectively, disk cache with a volume mount seems like a very reasonable solution. However, when the development only began there was still some uncertainty as to whether we want to use a microservice to serve statistical and machine learning models. For this reason we decided to go with Redis because it would fit nicely in a more complex architecture, and generally speaking it is a trusted battle-tested solution.

The next question is what should be cached. With R/Shiny most often the answer is to cache the outputs using the *bindCache* function. No matter how much data was required to prepare an output - Leaflet map or a Plotly chart - usually the output itself does not take much memory. And we need to keep the cached values small to make sure that the very purpose of caching is not defeated. Although, one thing that we discovered when testing the application is that whenever an error occurs inside the reactive that is being cached, Redis storage would be blown out of proportion. A cached entry that would normally take 100 KB can now take up to 10 MB. The consequence of this is that the memory limit in Redis will be reached too soon, so it will start overwriting old keys. Again, defeating the whole purpose of caching optimization.

USAGE STATISTICS

The collection process of usage statistics usually comes in three parts: instrumentation, storage, and visualization. Health Equity Explorer is instrumented with the *shiny.telemetry package* to ensure that every navigation and every input change is registered. All telemetry logs are sent to a utility database - Postgres - that is launched together with the Shinyproxy container. This database is connected to another Shiny application - analytics dashboard that comes out of the box with *shiny.telemetry*.

APPLICATION DEEP-DIVE

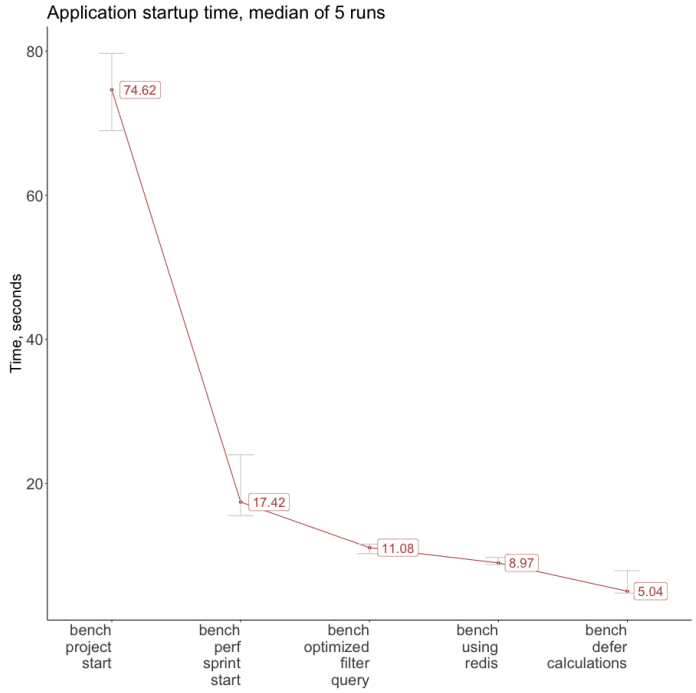
Next we will discuss some of the technical highlights of the Health Equity Explorer - R/Shiny application - itself. In this paper we discuss neither basic R/Shiny development, nor Rhino specific practices for there is plenty of materials available online on these topics. However it should be noted that Rhino has played a critical role in the development of every feature discussed below - either making it possible, or making it substantially more maintainable and easy to implement.

PROFILING, BENCHMARKING AND OPTIMIZATION

When deploying Shiny applications using Shinyproxy, a great deal of attention should be dedicated to application startup times. To achieve the best possible results, we instrument the application with logs and start profiling everything - function execution, module sourcing, library loading - using the *tictoc* package as well as Rhino's *log* module. This process alone allows us to identify a few problems. One of the greatest examples is loading the *sf* package. It takes 5 to 6 seconds to load this package, and it is required to read geospatial data and render the interactive maps. However, to get to the maps user first needs to go to the "Neighborhood Data" section of the application. In other words, loading this library eagerly is an expensive but unnecessary task. The solution is to load required sf functions using the *box* package only inside the shiny module where those functions are actually used.

To ensure that the optimizations we implement have real impact on the application, we set up benchmarking using *shiny.benchmark* and *shinytest2* libraries to track application startup time. At first, the benchmarking is done locally to fix the biggest problems on a large dataset. When application loading becomes independent of the database size, we can move benchmarking into the CI and make sure that we have this metric under control with every new pull request. The figure below demonstrates the benchmark results throughout several pull requests dedicated to startup time optimization. Each point on the chart demonstrates an average, minimum and maximum startup duration across five attempts.

Fig. 2. Startup time optimization chart

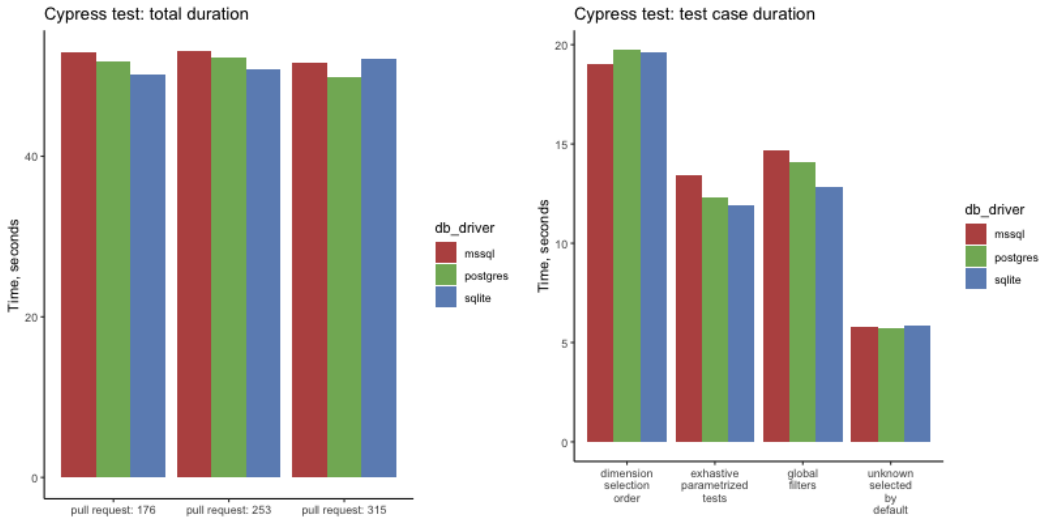


As you can see, when we started working on the project it took on average almost 75 seconds before the app was loaded and fully interactive. The biggest performance optimization that resulted in 17 seconds load time is the outcome of our first "performance optimization sprint". During this time we removed some eager data fetching, repeated calculations, optimized namespace loading (the notorious *sf* and *terra*, but also some application logic modules). Next improvement down to 11 seconds is related to rewritten SQL queries. Avoiding ORM and query generation tools like *dbplyr* for certain mission-critical tasks is an overlooked, but very powerful optimization technique. When we introduced Redis you can see that in terms of startup time it gave a somewhat marginal improvement of about 2 seconds - this is due to the fact that we were now able to avoid some repeated fetching. Final substantial improvement is related to deferred data fetching and processing - we refactored the code in a way that as little data as possible is fetched/computed outside of the *shiny server* function. The final result is an astonishing 15x faster application loading.

After startup time becomes satisfactory, we shift focus to intra-session benchmarking. Health Equity Explorer features a lot of data-intensive computations while also supporting multiple database drivers: SQLite, MS SQL Server, and PostgreSQL. To make sure that the code works adequately across all three and maintains reasonable execution time,

end-to-end tests are used. We use e2e testing in two ways: (1) simulate user behaviour and assert the expected outputs and results, (2) measure running time for each tested scenario. While uncommon, it is not too difficult to use Rhino's default e2e testing engine - *Cypress* - for benchmarking. The recipe is as follows: use "teamcity" reporter in Cypress configuration, save the output of *rhino::test_e2e()* into a log file, process the log using R to extract teamcity lines with timestamps and calculate running time, save the results in a csv file, and visualize on a *ggplot2* chart. The best part about this process is that we do this automatically in Github Actions for every pull request that is merged into the main branch. The figure below shows the output of this process that is saved in the github repository.

Figs. 3-4. Performance benchmarks across various databases: total vs case-by-case duration



CUSTOM REACT COMPONENT

From backend optimizations we slowly move towards the optimizations on the frontend. Our expertise in web development has allowed us to enhance user experience by keeping expensive frontend tasks on the client side using React.js. Normally, developers would go for React to add some non-trivial or extra fancy-looking components to their Shiny applications. While this holds true in our case, the actual reason to write some Javascript code is again related to performance and the requirement to scale the UI to support many more health equity dimensions.

Here is a user story: "As a user I want to analyze a health outcome dimension stratified by up to three attributes present in the data…". This doesn't sound too bad until we learn that there are hundreds of these attributes organized in a tree-like hierarchical structure with varying levels. Some of the attributes are categorical, but most are numeric - and the numeric ones need to be categorized based on the user input, i.e. on demand. For example, average household income in the neighborhood is a continuous variable and can't be used for stratification directly. So the user should be able to choose from a few options on how to categorize this variable first: 50/50, quartiles, deciles, etc, so that afterwards they can compare a certain health outcome across different income groups.

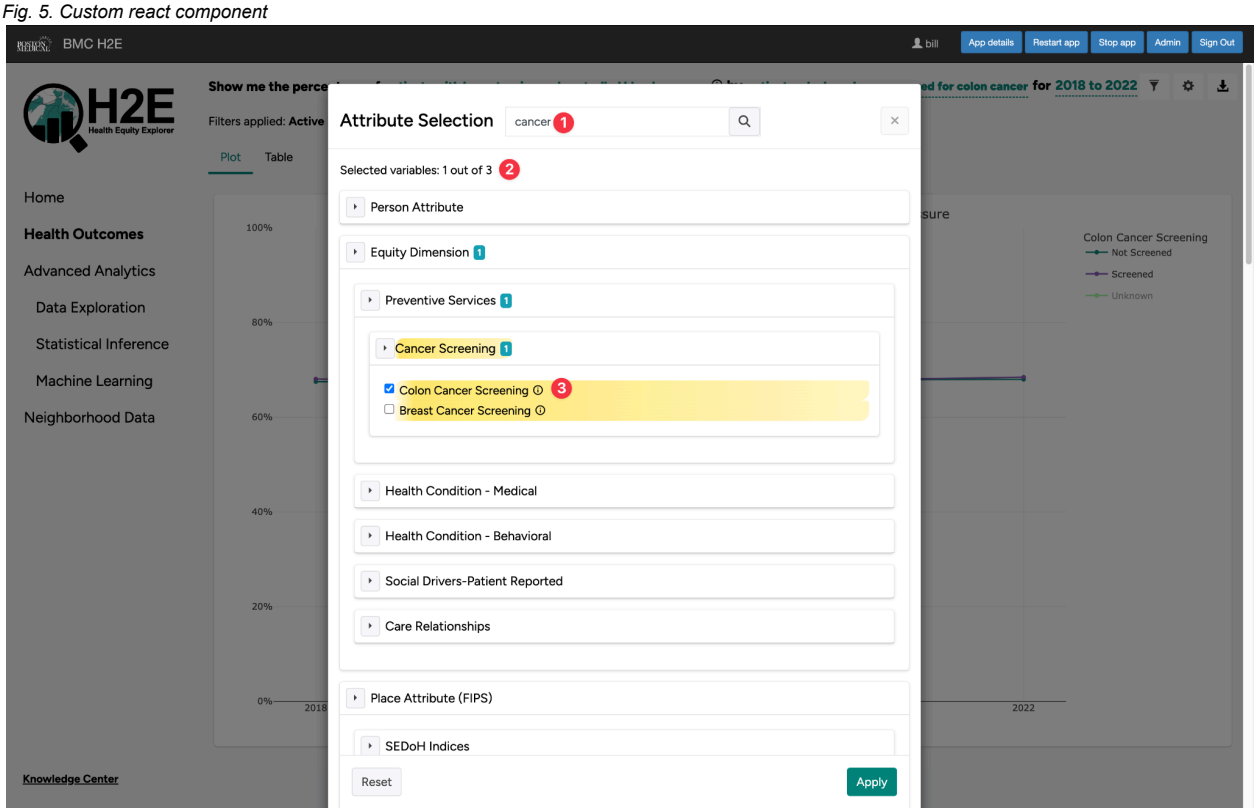
After some further refinement it was agreed that a custom component is needed that will satisfy the following criteria: (1) present different types of inputs (checkboxes for categorical variables and dropdowns with categorization options for numeric attributes), (2) display the attributes in a hierarchical tree form to represent the underlying structure, (3) be able to render a lot of inputs at once, (4) provide ability to search for a desired attribute, (5) limit user selection to only *X* number of attributes.

At first, we tried to solve this problem by assembling a new component from the existing Shiny toolbox - checkbox and select inputs. However it has quickly become clear that the frontend performance is a blocking issue here: when rendering more than *seven hundred* inputs Shiny would bind every one (input binding is what makes the communication between the web browser and the R process possible). The binding process would freeze the entire application for several seconds resulting in a poor user experience - especially considering the fact that this component is reused throughout the entire application multiple times.

Rhino integrates with custom React components so easily, that we decided to implement this feature using Javascript. We created a reusable component that is bound to Shiny only once, keeps its own state to limit selection and display currently selected attributes, and only sends a message to Shiny upon submission. It opens instantly, renders all attributes at once, allows users to search and highlight the needed ones. Of course the R counterpart of the component is necessary to define a Shiny Binding and a function used in the R/Shiny application that controls which attributes to render, and how.

Below is a figure demonstrating one of the uses of the component in the application with the following elements: (1) search query, (2) indication of currently selected attributes and the maximum selection limit, (3) attributes that satisfy the search criteria are highlighted and their containing sections are expanded. Also notice how the tree structure is clearly represented by the collapsible elements.

Fig. 5. Custom react component



MACHINE LEARNING FRAMEWORK

When thinking of machine learning modules that could be used in the application, there are two limiting factors: (1) it is not possible to fit any meaningful model on synthetic data, (2) it is difficult to anticipate which models will be required at the organization where the application will be deployed.

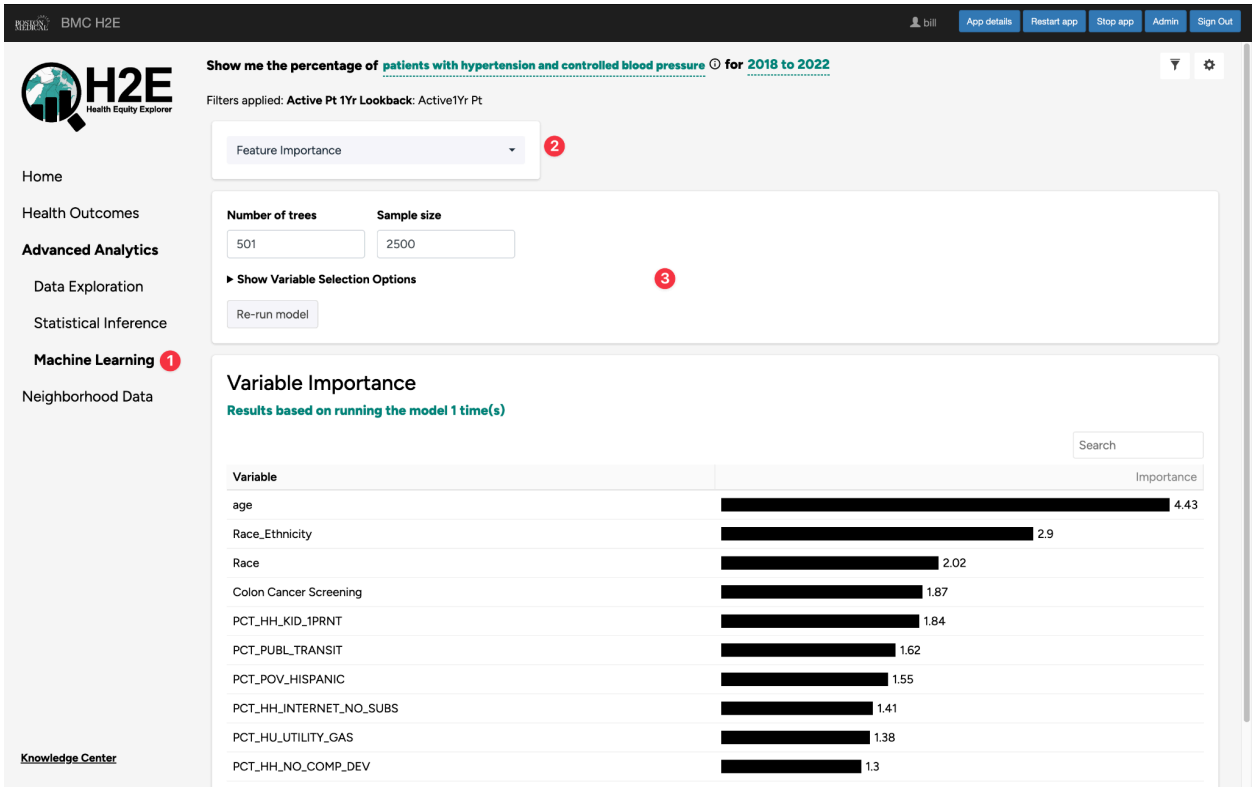
For this reason, we developed a so-called “Machine Learning Framework” - a system of Shiny Modules built around Rhino’s integration with the *box* package. In other words, each ML model is a mini Shiny application with its own UI and Server that are nicely incorporated into the main application. Every mini-app receives a pre-defined set of inputs - selected health outcome, dimensions, patient records - while also having access to the global Shiny state stored in *session$userData* object. Apart from the R module itself, authors are able to accompany it with a piece of documentation written in a markdown file. Thus, every machine learning module consists of two files: module.R and module.md.

It all comes to life thanks to a Module Factory that we created specifically to facilitate this process: it reads the content of machine learning modules directory, dynamically creates shiny modules and adds them to the main application where these modules are available in a dropdown selection. It also reads module documentation from a markdown file and adds it to the special “About” section. The developer of each module is able to control how the module appears in the dropdown by exporting a special *LABEL* variable, and the application administrator can control which modules should be hidden via a config file.

As noted earlier, it is quite challenging to develop useful models in advance, but we nevertheless ship the project with one machine learning module - primarily to provide future contributors with a tangible working example. Such an example must have a good balance between simplicity and complexity: it should showcase all features required for a module development, preferably bring value to the users, while keeping things relatively isolated. Taking all these factors into consideration, we developed a feature selection module using random forests algorithm - arguably one of the few ML tasks that is relatively data-agnostic. Even though it was developed using synthetic data, it should also work when the application is connected to the real patient data. Most importantly, we extensively described the process of module creation in the project documentation - how the module should be written, where to put files, which extra configurations to use, etc. The figure below demonstrates how the Machine Learning Framework looks like in

the Health Equity Explorer application with the following items: (1) ML section of the application, (2) dropdown with available machine learning modules, (3) body of a module.

Fig. 6. Machine learning module example



CONCLUSION

This paper describes the technical highlights of the Health Equity Explorer. We touched on the system design, and why it is so important in complex applications like this. We then discussed in detail several features of the main application that make it stand out compared to most applications.

REFERENCES

1. Adams WG, Gasman S, Beccia AL, and Fuentes L. The Health Equity Explorer: An open-source resource for distributed health equity visualization and research across common data models. Journal of Clinical and Translational Science 8: e72, 1–9. doi: 10.1017/cts.2024.500
2. OMOP Common Data Model. https://ohdsi.github.io/CommonDataModel/. Accessed February 10, 2025.
3. Shiny Redis Cache implementation. https://shiny.posit.co/r/articles/improve/caching/. Accessed February 10, 2025.

ACKNOWLEDGMENTS

The authors would like to express their deep gratitude to all the people involved in the development process of the Health Equity Explorer at various stages and times: Michał Parkoła, Dawid Strytyński, Jakub Stepniak, Ege Can Taşlıçukur, Shankhadeep Dutta, Arkadiusz Kalandyk from Appsilon; Sarah Gasman, Shubham Chakankar, Mike Mendis from Boston Medical Center, as well as everyone else who helped along the way.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Author Name: Pavel Demin
Company: Appsilon
Address: -
Work Phone: -
Email: pavel.demin@appsilon.com
Website: https://www.appsilon.com