mediTree – an interactive exploration on decision trees

Arwed Walke[*] ETH Zürich
Ben Martin[†] ETH Zürich
Colin Rüegg[‡] ETH Zürich
Deborah Zanette[§] ETH Zürich

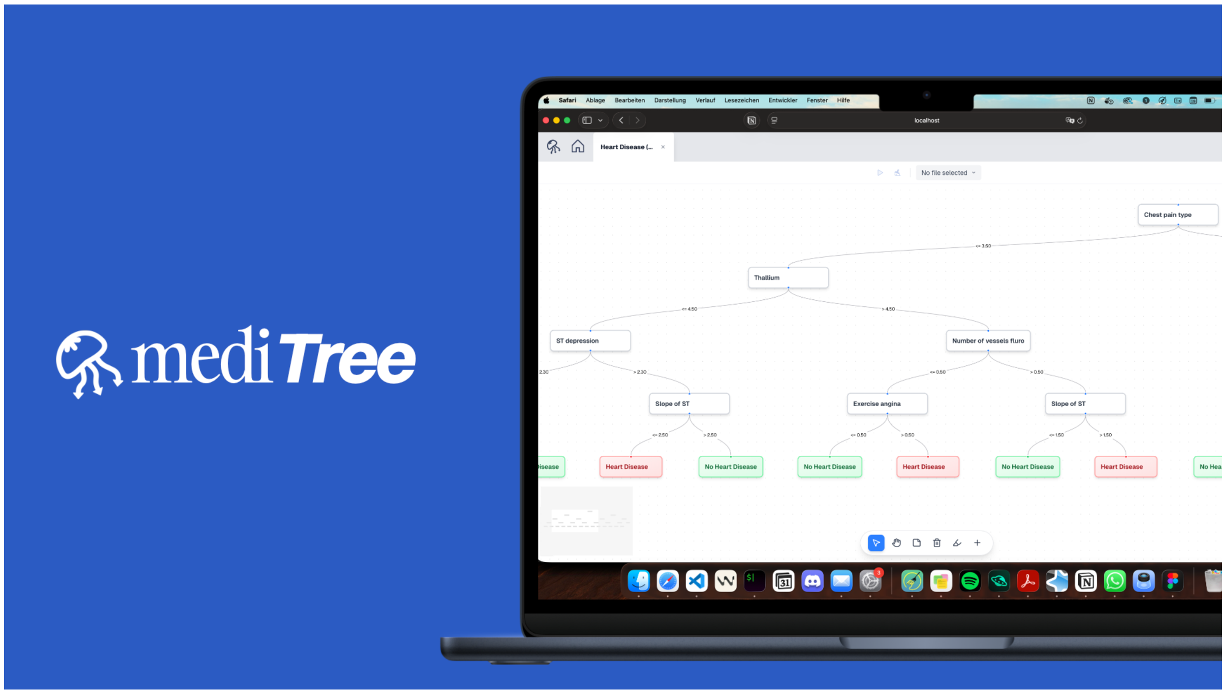


Figure 1: mediTree graph view teaser image.

Abstract

mediTree provides a structured tool to experiment with automatically-generated decision trees. Based on a given dataset, the prediction of important information such as ICU stay or heart disease probability are useful tools for hospitals, but especially on limited training data, trees can overly emphasize special cases or completely disregard patients with characteristics not contained in the dataset. The idea of meditree is to give doctors full control over the decision trees, which are pre-generated automatically.

1 Problem Outline

Computerized clinical decision support systems have been used in medical fields since the 1980s. They help clinicans with a number of tasks.[1] In our case, we developed a tool which helps doctors diagnose patients with heart disease. The task was to implement a visualization and editing platform for machine-generated decision trees. Given a tree of depth n fitted on a dataset, the goal was to expose editing and correction features to domain experts. In our case, the tree was supposed to be trained on a patient dataset to predict ICU stay duration, but we pivoted to heart disease prediction later since better data was available. We also implemented the tree creation itself, since this functionality fitted the microservice requirements really nicely.

[*]e-mail: awalke@student.ethz.ch
[†]e-mail: benmartin@student.ethz.ch
[‡]e-mail: rueeggco@student.ethz.ch
[§]e-mail: dzanette@student.ethz.ch

2 Implementation Process

The work split was rather clear early on. Ben coded the frontend, Arwed took care of the backend, database and CI/CD, Colin programmed the Python microservice and Deborah worked on UI and backend routes.

Once we understood the desired deliverable, we had some very clear ideas and got started setting up a Notion page for project management, task tracking, design inspiration and notes.

2.1 Brainstorming

Our key considerations were:

- How do we visualize the tree and make editing feel intuitive, even at high complexity?
- How can we build the project in a way that enables secure storage of potentially sensitive data?
- How do we handle file storage and collaboration so the program gets out of the user's way as much as possible?

We drew the conclusion that React Flow was going to power the editor, and wanted to use PocketBase for the data storage, since it was easy to self-host and a breeze to configure and hook up using the extensively documented API. This improved our DX since we got around writing SQL queries and cleared out any privacy concerns right off the bat. After the second coursework assignment, for which we implemented a backend that stores files that could be uploaded from several clients, we noticed two things. Firstly: the possibility of multiple people editing the tree, such as researchers working remotely, doctors tweaking the trees, and possibly other people requires some form of synchronization protocol. Secondly: simply recreating the backend from the second assignment again for the final project would, frankly, not be a very impressive feat. For this reason, we really wanted to add collaboration to the project, in the style of Figma, Google Docs and the likes. After some research, we concluded that integrating React Flow with the Y.js library for collaboration would be feasible.

2.2 The Mock-up

Our first step towards tackling the project was to create a Figma project and coming up with a color palette. We associate the medical field with a more cool-toned and sterile look, so we went with shades of blue and gray. Then we made a first mock-up of a tree and an information pop-up, which was ultimately scrapped and integrated into the side panel. We also came up with a design for the tree editor and later one for the home screen where you get an overview of your trees. The main focus was on making a minimalist design that is easy and intuitive to use, while also retaining all relevant functionality. Since we were originally supposed to predict ICU stay duration, we focused on making an easy to use interface that would not be an obstacle in a stressful and fast paced environment like the ICU. This design paradigm is also important in other less time sensitive environments in the medical field, so we didn't have to make any UI adjustments when we switched to heart disease instead.

2.3 Coding

Before we address the programming process, we provide a short overview of the technical dimensions of our product to help understand our process.

2.3.1 Tech Stack

We ultimately decided on the following technologies to approach the implementation:

- Next.js + TypeScript for the frontend, graph powered by React Flow
- TypeScript backend (REST API) using Hono
- Y.js to power live collaboration
- Socket.io as a glue between client and server
- Pocketbase as a light database and file storage
- Hoppscotch for API and backend testing
- Docker containers for each app component
- Python + FastAPI microservice
- Decision tree fitting using scikit-learn

2.3.2 Technical overview

Our backend can be understood as a hybrid HTTP/WebSocket solution that enables basic CRUD operations on all saved tree files via a REST API. Whenever a user actually wants to edit a file (i.e. when you click on a tree in the file explorer), the backend loads the tree from disk into an in-memory document map. The socket.io subservice maintains a room and an associated Y.js document for each document with active users and whenever changes are made to the nodes, edges, values or any other file metadata, clients send an update message via socket.io. The server then reconciles all changes and distributes them to other clients using Y.js's conflict-free replicated data type (CRDT) system.

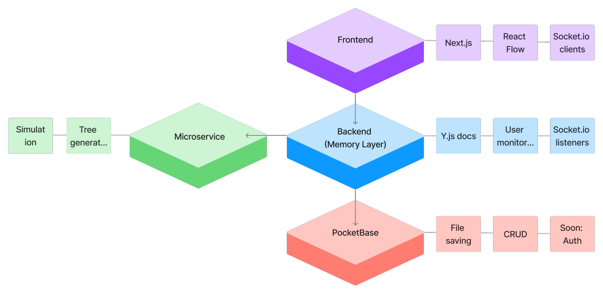


Figure 2: Technical visualization

Whenever a room is empty (which happens when all active users have closed the tab or disconnected), the affected document is saved to PocketBase and removed from the in-memory document map to save performance. We also run a routine that regularly checks for inactive sockets and backs up all active graphs.

Users can create blank trees, copy existing ones and fit trees on uploaded data. The last two options trigger a short socket.io negotiation, in which the backend clones the requested file or requests a fitted tree from the microservice, and notifies the frontend when it is ready to be loaded.

2.3.3 Kickstarting backend and node editor

The first lines of code we wrote addressed the backend and the node editor. While Ben prepared the node editing, Arwed started preparing the infrastructure necessary for tree saving and Y.js collaboration. This involved agreeing on a data scheme that both sides of the application could understand. Furthermore, the node editor was not programmed to support collaboration out of the box, so Arwed wrote a lengthy React hook to set up a persistent socket instance whenever the tree is loaded. It automatically registers with the corresponding room when the user opens a document, and disconnects from the room upon destruction. After a bumpy integration sprint and a code issue that caused the tree to disappear after some edits due to React's page rendering logic, the hardest part was done: node editing was collaborative, and synced to the in-memory YDocs.

2.3.4 Backing up decision trees

After some iterating and testing, implementing file saving in addition to the in-memory system we had in place became increasingly inevitable. This required properly dockerizing PocketBase in order to run it on other teammates' machines and in the deployment, as well as setting up the Hono endpoints. This turned out to be quite the struggle, because the modularization Hono supposedly offered repeatedly failed us for no apparent reasons. Originally, we planned on having a separate file with a Hono instance for each major API function, with a top-level instance that routes to the other routes, as described in the docs. After a lot of debugging, we had to opt for a single instance with subroutes instead to make the endpoints work consistently. After that, it became clear that the current socket.io system for files would not take us far enough. The first version opened files by their name instead of their unique IDs and created files in case the requested name was not found. But this did not allow granular control over the file type (for instance, creating a blank tree, forking an existing tree and requesting a pre-computed decision tree from the microservice all require wildly different control flows in the backend). Hence, we went back to the drawing board to figure out a better message protocol for file creation and revamped the ID system. After ensuring that everyone could actually log into their local PocketBase instance and establishing a consistent database scheme, the groundwork was laid.

From there on, we only had to implement moderately sophisticated auto-backup and cleanup logic, which started working quickly after the first commits.

2.3.5 The Microservice

The microservice acts as the computational engine of the application, built using Python and FastAPI. We chose this stack to leverage Python's extensive ecosystem for data science, specifically 'scikit-learn' for machine learning tasks and 'pandas' for data manipulation. The service handles two critical functions that require a bit more computation: generating the initial decision trees and running patient simulations.

When a user requests a new tree fitted on data, the microservice trains a Decision Tree Classifier on the heart disease dataset. A notable detail of our implementation is that the service performs the graph layout calculation server-side. It traverses the 'scikit-learn' tree structure and maps it directly to React Flow nodes and edges, computing the X and Y coordinates for a tidy visualization. This ensures that the frontend receives a ready-to-render graph, offloading the complexity of tree layout algorithms from the client.

The second major component is the simulation logic. To support the workflow, we needed a way to test the user-edited trees against real patient data. The microservice accepts a batch of patient records and the current graph structure, then simulates the path of each patient through the tree. It parses the conditions on the edges—handling boolean logic, numeric comparisons, and thresholds to determine the final classification for each case. This also allows the application to provide immediate feedback on the accuracy of the doctor's modifications on the training data set.

2.3.6 Deploying PocketBase

One of the most time-consuming and exhausting parts of the project was setting up PocketBase to run in the backend. This was especially difficult in the beginning, where Arwed had put into place a makeshift solution for which the PocketBase executable was downloaded into the backend container, with both sharing the same Dockerfile, since exposing ports on the deployment without assigning specific domains did not work well. It was only in the last week that we realized we could just replace the unused Postgres Dockerfile with PocketBase, but even then, configuring a default superuser and testing the deployment turned out to be quite a hassle, since we had to wait for 90% of the deployment to finish every time we changed the pipeline before seeing results. Even worse, since we used a docker compose setup locally, the Helm configuration was notoriously difficult to test outside of the GitLab environment.

The trouble eventually culminated in a crashed helm release that blocked the build pipeline until we manually managed to take it down. But after that, the database finally ran on its own subdomain and we could connect all app parts and URLs with little to no obstacles.

3 The UI

As it was already outlined in the mock-up section, our main objective was to make the website easy and intuitive to pick up, since doctors are busy and don't have time to learn to use a complicated new tool. Our goal is to provide a practical tool that assists them in diagnosing patients. If the tool is too much of a bother to use than to just get a second opinion or to thoroughly go through the data yourself, then it's pointless. But having a working tool is only half of the equation, which is why we put a lot of thought into the UI and striking a balance of functionality and simplicity.

3.1 The Homepage

The Homepage provides a neat overview of already existing trees and the option to create a new tree. Besides starting a blank tree, you further have the option to start with a decision tree of specified depth fitted on our training data to predict heart disease. The header allows you to seamlessly switch between tabs which contain your trees or to go back to the homepage. The main area of the homepage shows all of the existing trees with their names, so a doctor can quickly find the tree they're looking for.

When users open the app for the first time, a short welcome page explains the workflow and features for a quick onboarding.

3.2 The Tree Editor

The main challenge of the frontend work was building a tree component that fulfilled all of these requirements:

1. The tree supports medical staff in making patient decisions on empirical data while still giving final authority to the person in power.

2. Medical staff should easily edit decision trees based on present facilities and domain expertise. Doctors can easily change values, add notes, change, add or delete nodes in the tree

3. Ease of use and an intuitive user interface has to be guaranteed in every part of the product to not add additional mental burden to medical staff in stressful and time-critical situations

4. The tree visualizes and simulates patient histories where doctors can easily enter and edit patient data.

5. Evaluate the accuracy of a selected tree based on selected depth and custom edits done by medical staff.

3.2.1 Tree Editing

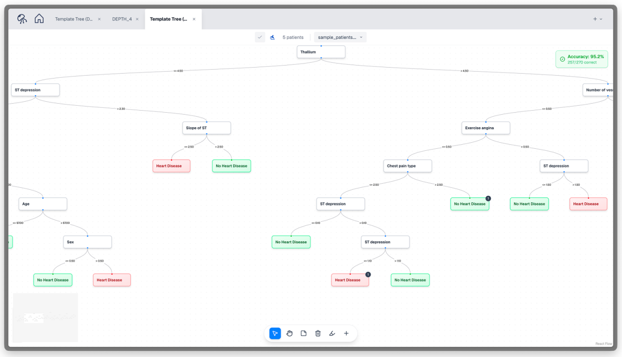


Figure 3: The tree component in action

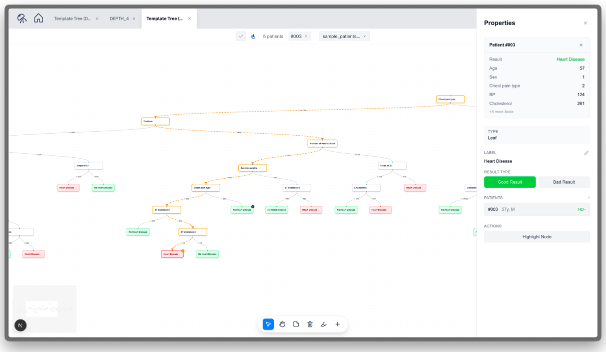
We decided to make the main area of the tree editor blank, so the user can fully focus on the data. A toolbar offers all of the necessary tools; select, pan to move around, a sticky note feature to comment on nodes, thresholds or even the whole tree, a highlighter to mark especially important components and a node and leaf adder. There is also a small map in the bottom left corner to give an overview of the scale of the tree. When more information or advanced edits are needed, you can select a node and open the side panel, which lets you change the feature of the node or adjust its thresholds.

3.2.2 Nodes and the Sidepanel

In the tree there are 2 different types of nodes: decision nodes that can have ingoing and outgoing edges. These decision nodes are used to classify patients into distinct subtrees where the classification is based on medical features such as age, blood pressure, heart rate or certain blood markers.
The second type of node is a leaf node that ends the path for a patient and puts the patient into a bucket, in our current demo these buckets are heart disease and no heart disease based on our training data. However, our infrastructure also supports other types of classifications through all layers of out tech stack. In the future, this infrastructure could be used to predict features such as length of ICU stay, necessity of certain medical interventions such as the need for mechanical ventilation, etc.

3.2.3 Patient Simulation

Our tree supports uploading a .csv or .xlsx file with patient data where each row contains the relevant patient information. If this table fits the attributes of our tree, you can choose to simulate this table directly on the tree. When you choose to simulate by clicking the play button below the header, all patients in the file get added to their respective leaf nodes. By choosing a leaf node then, you can get an overview of patients in this bucket, view the attributes of these patients in the buckets, edit the patient data and view their path through the tree to their final leaf node. Figure 4 shows such a highlighted path for a sample patient from an uploaded table.



Result	Heart Disease
Age	57
Sex	1
Chest pain type	2
BP	124
Cholesterol	261

+8 more fields

TYPE: Leaf

LABEL: Heart Disease

PATIENTS 1: #003 57y, M HD+

Figure 4: The simulated path of a patient and the side panel on the right

4 What we learned

The final project was quite a step up from the previous assignments, which made it a lot more interesting to work on. For some of us, this was the first full-scale project they have built since the start of our studies, and it was impressive to see how much more capable we have become since.

The final version of the app also closely resembles the Figma mockup because we did not have to iterate too much on it, which is a success in our opinion.

We also did not have prior experience with machine learning, so we really appreciated the opportunity to learn something new and have it integrated into a real, tangible application right away.

Some notes on vibe coding: While using AI agents in cursor and using LLMs definitely sped up our development, current models are not yet at the point that web development can be considered a solved field. All architectural decisions had to be done completely by hand, appropriate libraries, bug fixing, review still has to be done by hand to avoid hard to detect and debug errors. Although it allowed us to make rapid progress at first, AI was more of an advanced auto-complete service than an additional team member to us.

On the topic of infrastructure, this project definitely taught us a lesson on the topic of monolithic repositories. Although many successful projects are built on monorepos, we quickly realized that with an increasing complexity, the single build pipeline that encompassed all layers of the tech stack bottlenecked our workflow. Simple changes to the database configuration required a 15-minute wait time to test in production, whereas the local docker-compose build only took a couple of minutes. This proved especially challenging when we noticed a bad CORS configuration, causing the microservice to deny requests, which could not be debugged locally due to all services having the same localhost domain. Having individual containers and workflows for each component would have improved our developer experience and iteration speed significantly.

5 Limitations & Next Steps

While our current platform provides a valuable contribution in utilizing decision trees for prediction tasks, it is primarily constrained by available data. Our infrastructure supports creating and working with decision trees for many more clinical decisions beyond predicting heart disease risk, but we have not been able to find reliable, cleaned datasets for these additional use cases. A first step towards making these decision trees more broadly usable would be to build data sets which they could be fitted on. To transition from a prototype to an indispensable tool for medical researchers and clinicians, our infrastructure needs to generalize across a broader range of tasks. This requires expanding our data pipeline and demonstrating the system's capabilities across multiple clinical prediction scenarios. Additionally, we have identified distinct user requirements that suggest splitting our product into two specialized versions: one for clinicians and one for researchers. Clinicians working in intensive care units require extensive documentation capabilities, seamless integration with local hospital equipment, and streamlined workflows optimized for bedside decision-making. Researchers, by contrast, need deeper control over the machine learning components, including the ability to create custom test sets, access advanced data analysis tools, perform model comparisons, and fine-tune algorithmic parameters. By developing these as separate products with tailored interfaces while maintaining a shared core infrastructure, we can better serve both communities and maximize the platform's clinical and research impact.

6 Conclusion

It's fascinating to see how closely two fields like programming and medicine are intertwined and how much ML can contribute. We constantly make decisions, but when making life or death decisions in stressful environments it's certainly helpful to have some assistance. Often one can overlook trivial details or forget them entirely. While a decision tree model couldn't replace a professional, such as a doctor, clinician or researcher, it can certainly be a tool to understand the data better and get a visualization of very theoretical concepts and metrics. The field of ML trained decision trees is promising and certainly a valuable tool to consider using in complex decision-making processes.

Acknowledgments

We would like to thank Diana Robinson and Robin Chan for supervising our project.

References

[1] R. T. Sutton, D. Pincock, D. C. Baumgart, D. C. Sadowski, R. N. Fedorak, and K. I. Kroeker. An overview of clinical decision support systems: benefits, risks, and strategies for success. *npj Digital Medicine*, 3:17, 2020. doi: 10.1038/s41746-020-0221-y 1