AutoHopts Collaboration with DeepCure

Team Members

Daniel Belmes, Ira Ceka, Kristen Laird, James Michaud, Rudresh Nitinku

Functionality

The goal of our app is to make it faster and easier for DeepCure employees to start new hyperparameter optimization experiments. We accomplish this by creating a web app that users can use instead of using the command line. Our app is designed to support users with three different experience levels: beginner, intermediate, and advanced.

The top section of the app—above the dropdowns—is the only required part. The user must specify basic information like experiment name. They can then submit the experiment; everything else is set to default values. This reduces the complexity of their task and makes it easier for a new employee to get up to speed. DeepCure administrators can modify default values in the backend, allowing them to guide other users toward making good decisions. Validation prevents users from submitting if all fields have not been filled out.

An intermediate user with more knowledge might choose to interact with the dropdowns. Changing the problem type and model type in the top section changes the models that appear in the models dropdown. Checking and unchecking individual models determines which models will be included in the experiment. If all models are unchecked, an error message appears on the dropdown to indicate that at least one must be selected. Each individual model has several parameters associated with it of three possible types. There are multiselect categorical parameters, float/ints with quantization, and float/ints without quantization. These are populated with default values that DeepCure sets, as well as validation values that DeepCure can determine. For example, at least one value in a multiselect must be chosen, and lower must be less than upper as well as within bounds determined by DeepCure. If the user collapses an individual model, a count of errors is displayed next to that model. If the user collapses the entire models dropdown, a count of errors over all models is displayed on the dropdown, making it easy to see errors even when sections are collapsed. Intermediate users can similarly choose to edit the other dropdowns, which contain flags for configuring different parts of the experiment.

Advanced users who are extremely confident in navigating experiment setup can choose to upload their own flag or model configuration templates. These override the default values and automatically populate corresponding fields. Because advanced users typically have one or two templates that they work from as the basis of an experiment and then they refine the experiment from there, uploading does not change which models or parameter values are selected. The user still needs to select the appropriate values in the dropdown, but this is intended functionality because it allows those advanced users to have a single starting template for experiments that they work from. During upload, the app verifies that files are formatted appropriately and throws an error if the user tries to upload model names, parameter names, and flag names that do not exist in the default values to ensure the experiment will be runnable.

Ones the user presses the submit button on the main page, the confirmation page displays the two files that are created. One shows model configurations, and the other shows flags. It is formatted in the same way as

the JSON inserted into the database in the next step, allowing DeepCure employees to verify that it looks appropriate. Once the user confirms the experiment setup, it writes to the database, switches back to the main page, and displays either a success or failure notification. The previous values are intentionally not cleared so that an employee can quickly start another experiment very similar to the one they just submitted without having to go through the same setup work. If they would like to go back to the default values, using the browser refresh button resets everything.

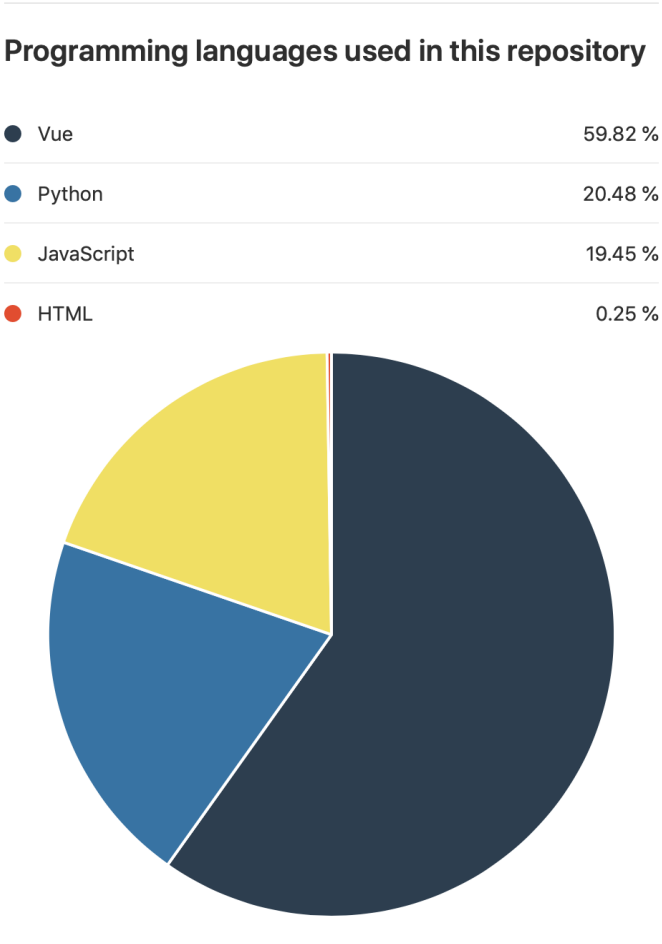
The app in its current form is ready for DeepCure to take the next steps of hosting the site, replacing our test database with their real database, and connecting to their job scheduler, which was the defined MVP for our project.

Code Structure

We used a VueJS tutorial to kick off our project, which left us with a basic code structure to build upon. At the root directory we broke the project up into three main folders: client, server, and example_upload_files. example_upload_files is self explanatory. server contains the backend python files along with the default configurations files.

The client folder contains our frontend. There are the basic files and folders created by NPM that are used to build the app along with a src folder which contains the majority of our work on the frontend. Below is a directory tree which illustrates our code structure, emphasizing the folders containing our work. The below pie chart provides a breakdown of the coding languages we used. The Vue slice consists of source code with a .vue extension. These files are made up of a combination of HTML, Javascript, and CSS.

```
├── client
│   ├── src
│   │   ├── assets
│   │   │   └── [custom images and logo]
│   │   ├── components
│   │   │   ├── Main.vue
│   │   │   └── [other custom components]
│   │   ├── router
│   │   │   └── index.js
│   │   ├── App.vue
│   │   ├── event-bus.js
│   │   └── main.js
│   ├── [node build and configuration folders]
│   └── [node build and configuration files]
├── example_upload_files
│   ├── all_configs_new_values.json
│   └── all_flags.json
├── server
│   ├── run.py
│   ├── default_configurations.json
│   └── [other .json and .py files]
```



Tools and Libraries

GitLab is a lifecycle tool that provides a Git-repository manager

We used GitLab to collaborate, host our source code, push and merge changes, and track issues and milestones

Vue.js is an open-source Javascript framework for building user interfaces and applications

Vue was the main tool in building our application

Python is a popular high-level programming language

We used Python to support back-end work and RESTful API operations

npm is a package manager for the Javascript programming language

We used npm to download tools and run our app

Flask is a powerful micro web framework, useful for building RESTful APIs

We used Flask to power our backend RESTful APIs

Full list of packages used and their versions:

Package	Version
Click	7.0
Flask	1.0.2
Flask-CLI	0.4.0
Flask-Cors	3.0.7
Flask-SQLAlchemy	2.3.2
Jinja2	2.10
MarkupSafe	1.1.1
SQLAlchemy	1.3.1
Werkzeug	0.15.1
flask-marshmallow	0.10.0
itsdangerous	1.1.0
marshmallow	2.19.2
marshmallow-sqlalchemy	0.16.1
npm	0.1.1
optional-django	0.1.0
pip	19.1.1
psycopg2	2.8.2
setuptools	39.1.0
six	1.12.0
xxhash	1.3.0