From Data Science to Neuroscience
Making the Brain Visual and Accessible

A Division III Project by Brooke Fitzgerald
Motivation

Navigating a Landscape of Technology

The World Wide Web

How is a Web Page accessed?
What Makes a Web Page?
Data and the Web - Ajax and REST APIs

Web Framework

Why Django?
Django's Structure

Models, Templates, Views

Model
Template
View

Changing the Database - Migrations and SQL
Creating and Using a REST API with Django

Creating - Django Rest Framework
Using - Promises

Django and Security

Putting it all Together - Serving in Production

Visualization

D3
Graph-Scroll.js

Difficulties

How to Serve?
Data Size

From Tech to Product

Visualization Example: Percent Selective Over Time

Website Text

Wrapping Things Up

Works Cited
Motivation

Interactive and visual websites are becoming a more and more popular tool for communicating information, and for good reason! As news becomes more data driven, more graphics are needed to display said data. Interactive websites that utilize visualization, simulation, and play to communicate complicated concepts are being recognized as a better alternative to static textbooks (Victor 2011). The web is even being used to communicate new scientific research, as there are now entirely web-based scientific journals that leverage the web’s interactivity to explain and disseminate new research results more effectively. (Olah and Carter 2017)

With all of this in mind, I wanted to build a website that allowed people to ask questions and to gain a visual intuition and understanding of how neural data analysis works. Beyond knowing that I wanted to make a website, I also had to think about how my website would look.

In the interactive web development community, there is an approach to web-based narrative called scrollytelling. Scrollytelling is the idea that a website and the interactive graphics within that website should adjust and animate naturally with scrolling, as opposed to scrolling through multiple static graphics or forcing users to manually click or step through interactive graphics (Stolper et al. 2016). This approach allows you to easily couple data driven visuals and interaction with a natural interface for the user.

Because I wanted to create a story about neural data visualization (what it is, how to do it, what it looks like, etc.) accompanied by interactive graphics, scrollytelling seemed like a natural first step in designing my website. The concept of scrollytelling along with a few inspirational examples (http://www.r2d3.us/visual-intro-to-machine-learning-part-1/, https://mathisonian.github.io/trig/etymology/, https://pudding.cool/2018/02/waveforms/) gave me an idea of what I wanted my finished product to look like.

Once I had the idea for my website, I discuss in the “Navigating a Landscape of Technology” section the technology I could use to make my finished product happen and the struggles in doing so. Then in the “From Tech to Product” section I then talk about the final website - how one of the graphs is drawn, and include the text of the website.

Navigating a Landscape of Technology

When I started this project I had only tangentially worked with web technology. I had no idea how vast the collection of technologies used in every web application was, not to mention that they constantly evolve and expand.

In this section I will first discuss what makes a web page, how web pages are stored and requested, and how data gets sent over the web. I’ll then discuss the framework for my website
- how it works and how I set it up. Finally, I'll discuss how the visuals of my website are created, and challenges that I faced in creating the technical pieces of this project.

**The World Wide Web**

Using the internet seems a little bit magical - you type a string of letters into your web browser’s address bar and your web browser is transformed into a web page dictated by the letters you entered. For my thesis, I had to learn what it would take to make my own collection of web pages and make them available to anyone via the internet.

**How is a Web Page accessed?**

First, I had to figure out how a string of letters is translated into a web page. Each computer has a numeric location called an Internet Protocol (IP) address, similarly to how every business has a street address. Just like looking up business’ address by the business’ name, web browsers look up the IP address of a website by the **uniform resource locator (URL)**. When a URL like “https://www.google.com” or “https://neuraldata.net” is entered into the address bar of your web browser, your browser contacts a **name server** (the address book of the web) to find where the web page actually is stored. The name server receives the URL from the client, looks up the IP address associated with that URL, and sends the client the IP address. The client then sends a request to the computer with that IP address and the computer first returns a signal that it has received a request and will be sending the web page soon, followed by the web page. ("How Does the Internet Work?," n.d.)

A note about terminology: a **web page** is the document you see in your browser at one specific URL (e.g. [https://www.nytimes.com/topic/subject/computers-and-the-internet](https://www.nytimes.com/topic/subject/computers-and-the-internet)), and a **website** refers to a collection of related and connected web pages (e.g. all of the articles on [https://www.nytimes.com](https://www.nytimes.com)). ("What Is the Difference between Webpage, Website, Web Server, and Search Engine?," n.d.)

At this point in my research, the terminology became a bit confusing. When people refer to the **web server**, they are referring to either the computer that has the data, code, and applications that make up a website, or the application that sends (or serves) web pages over the internet. The web server that is the computer is also referred to as simply “the server.” For clarity’s sake, I will solely refer to the computer as the server and the application that sends web pages over the internet as the web server.
Figure 1: Diagram of what happens when a client requests a web page

The server stores all of a website's data and files and runs a web server. The web server sends that data to any number of web browsers that request it via Hypertext Transfer Protocol (HTTP). In addition to the web server, the server also often runs an application called a web framework or web framework application. The web framework structures the creation of a web pages to avoid unnecessary repetition, and provides the web server with dynamically generated web pages to send to computers that request them. The web framework can also inject data from a database dynamically into the web page. This is especially useful for displaying data conditionally, like if a user is logged in then the web page would display their account information or if a product is clicked then the web page would display the price and most recent reviews. (“Server-Side Web Frameworks,” n.d.)

The computer that makes a request of the server or web server is called the client. Every server can have multiple clients, and the client can send multiple requests to the server. A client can connect to the server and request the server to do computations. This type of client/server relationship is called connecting to a remote server. However, in the scope of this paper, the client/server relationship refers to the client making a request via a web browser to a web server for a web page sent over the internet.
With that in mind, there are two types of web servers. The type of web server that you interact with every time you access a public web page is a **production server**. This type of web server is an application that is continually waiting for a client to contact it and request the HTML, CSS, and Javascript that make up the web page of interest.

The other type of web server is a **local server** where the data and files on the server are sent to a certain **port** of the IP address 127.0.0.1 of that server (like 127.0.0.1:8000 or 127.0.0.1:5555). This is the type of server where tests are run and prototypes tested out before anyone with internet access can view the site (“How Do You Set up a Local Testing Server?” n.d.).

Note that both of these servers perform the same task - sending data and files to an IP address. However, the IP address of the local server cannot be accessed by any computer other than the computer running the local server. In fact, the name servers that store the links between website URLs and their IP addresses don’t allow any associations between a URL and the IP address 127.0.0.1.

After I understood how the server, web framework, and web server work together to send web page via HTTP to a client, I needed to figure out what exactly how to make the web page.

**What Makes a Web Page?**

Every web page can be broken up into three component parts: **HTML**, **CSS**, and **JavaScript**.

The skeleton of every web page is a **Hidden Text Markup Language (HTML)** document comprised of elements like paragraphs, links, and images (“Learning HTML: Guides and Tutorials” n.d.). The world wide web was initially intended to simply provide access to HTML pages. For this reason, if you’ve ever seen a sparse-looking website that is a bare-boned collection of text and images, it is likely just HTML.

For most, this austere way of presenting information is not enough. Thus enters **Cascading Style Sheets (CSS)**. CSS allows you to style your HTML elements by coloring text, shaping containers, and otherwise making your web page look nice.

While CSS makes the static HTML presentable, CSS and HTML alone cannot fulfill the dynamic, interactive, data-filled needs of most modern web pages. This is where **JavaScript** comes in. JavaScript is the only programming language that can run in all web browsers. The choice of JavaScript as the browser’s programming language is mostly for historical reasons, but there are programs that can take other languages (Python, Ruby, etc.) and compile them into JavaScript to use on the web. When a web page is loaded, JavaScript stores a nested representation of the elements in the HTML as well as their styling called the **Document Object Model (DOM)** (“Document Object Model (DOM)” n.d.). JavaScript allows you to dynamically
interact with the HTML elements in the DOM (animating a text piece on scroll, clicking an icon to reveal a dropdown menu), as well as send and retrieve data without reloading the entire page.

Data and the Web - Ajax and REST APIs

To send and retrieve data over the web, web page developers use JavaScript. The data that JavaScript sends and retrieves can come from a few places: a web server or a REpresentational State Transfer (REST or RESTful) Application Programming Interface (API).

JavaScript can access some data natively on the server, but only data that is sent over initially with the HTML and CSS of the requested website. In order to send data back to the server or get data on request without reloading the page, you need to execute Asynchronous JavaScript And XMLHttpRequest (Ajax) (Garrett 2005).

Ajax is the umbrella term for the collection of technologies needed to allow JavaScript to communicate with the server in the background (asynchronously) without needing to reload the page. XMLHttpRequest is the JavaScript function that sends HTTP requests to the server.

Keep in mind that the server doesn’t have to be the server that sent the JavaScript, CSS and HTML to the client in the first place. Any server that accepts ajax requests can send data to JavaScript, and servers handle these requests with a web application programming interface (web API). There are a few different standards for how data is sent and requested through a web API, but the most popular is a representational state transfer API (REST or RESTful API) (Fielding 2000).

A REST API is a way to expose application data over the internet to other programs, developers, and the like. This allows computers to access and manipulate application data in a predefined way. REST API is very important because it allows for developers to implement a few uniform methods that are the same across the internet.

In order to get data from a REST API, you need to make an HTTP request of the site in question with any other information you need. There are four main methods of a REST API: GET, PUT, POST, and DELETE.

To read data, you make a GET request. To create a resource in a database only if the resource doesn’t already exist, you make a POST request. To create a resource and overwrite the existing resource if it already exists, you make a PUT request. DELETE requests are used to delete resources.

When you make one of these requests you get two main responses. First, you get a status response that tells you whether or not your request was successful. Second, you get your data in a certain format. One format is XML, which stands for Extensible Markup Language.
where the “XMLHttpRequest” in “Asynchronous JavaScript And XMLHttpRequest” gets its name, but generally data is in the **JavaScript Object Notation (JSON)** format.

JSON data consists of key/value pairs, arrays, and other serializable values. As is hinted with the “JavaScript” in “JavaScript Object Notation,” JSON data is native to JavaScript and is easily converted into JavaScript objects.

**Web Framework**

Now, while a web server receives HTTP requests and returns HTML, CSS, and JavaScript to the client, a **web framework** dynamically generates the *specific* HTML, CSS and JavaScript that makes up the web page the client requested. There are many web frameworks available in almost any programming language (“Server-Side Web Frameworks,” n.d.).

**Why Django?**

Because I have a large amount of expertise in programming with Python, I decided to use one of the most popular Python Web frameworks - **Django**. Django is a “batteries included” web framework. It gives you a large amount of functionality right out of the box, and it scales very easily.

Django is what you use to send different pages to different URLs, structure your website’s data, filter and select data as necessary, and insert data into HTML, map URL requests to HTML pages, and to do computations on your data. Django also has a simplified syntax that manages receiving and sending HTTP requests through a web server.

The challenge with working with Django is that there is a fairly steep learning curve while figuring out how to leverage all of that functionality.

**Django’s Structure**

First, the overall structure of the framework. Within the Django framework, the **project** is your entire website, while an **app** is a contained piece of functionality. An example of a project-app relationship would be an event sales website (the project) that has an app to manage the users of the website, an app that handles ticket sales, and an app that manages the calendar of events.

Each project contains static assets (like images, fonts, JavaScript, and CSS), templates that provide basic HTML structure, and project wide settings.
Each app contains a file for testing your code, a file that links URLs with the correct HTTP response, files to make the app accessible by a site admin, and a file to register the app - essentially explicitly naming it so Django knows what to refer to the app as across the site.

Each app also contains files to enter models and views, and a folder for migrations, each of which is explained below.

Models, Templates, Views
For every app that you create using Django, you’ll rely on the Model/View/Controller software design pattern.

Each app within a Django project can be separated into model, template, and view components. The model handles your web application’s data, the view handles which data gets sent to each part of the website, and the template controls the data presentation. These components are what is called “loosely coupled,” which means that each component “knows” as little as possible about the other two. For example, the model layer can store data in a different way, and neither the view nor the template would change or break.

Model
The Django model is a way of specifying how data is stored in your database. Django’s model uses what is called a relational database. A relational database is a collection of tables that consist of rows and columns. A table generally refers to a distinct entity (like product or user) where the columns are general attributes of that entity (like price or first name) and the rows are specific instances of the entity (like a $14 wicker chair or Dan Carlin from Oregon). The names of tables and columns in tables are referred to collectively as the database schema.

Creating a model with Django takes classes and methods written in Python, translates them to SQL and creates tables from them in the database. Data is loaded into the database either through individual submission of data (either through form submissions or posting to a REST API) or by writing a script to do the initial loading. Django manages the state of the database with a series of scripts called migrations, which can do everything from editing existing columns or adding new database ones to running arbitrary python code.

Template
Every web page needs to have HTML and consistent visual components. Django provides a template system for dynamically creating HTML pages injected with data. Django templates look like basic HTML pages, but have a few more features.

First, Django templates can inherit properties from other templates. This makes it very simple to write a base page that contains the CSS, JavaScript, and other page elements that are consistent throughout the website. Second, Django templates can load in any page specific
CSS or Javascript. This allows for certain pages to have different styling or animations than the rest of the pages on the site. Third, Django templates can load data sent to the template and insert it into the HTML. However, the template itself doesn’t retrieve the data, and it will throw an error if the expected data isn’t sent to the template by the view.

View

The view is the intermediary between the model and the template. The view does two things. First, it is responsible for querying and filtering the necessary data from the database, then sending that data to a template to be visualized. Second, the view can convert the data from the database into JSON format (also known as serializing the data) and make it available as a REST API.

Changing the Database - Migrations and SQL

When the structure of the data underlying the web application changes, it is necessary to propagate those changes to the database schema. To facilitate this propagation, Django uses database migrations. A database migration is a way of outlining exactly what changes between one version of the database and the next. If you decide to add a new field to a Django model, Django creates a migration that adds the field to the right table in the database with a default value. If you create a new model, Django creates a migration to make a new table in the database. (“Migrations | Django Documentation | Django,” n.d.)

![Diagram of the different ways you can access data in the database with or without Django](image)

Changing the model causes Django to automatically create migrations that edit the database schema. Once migrations have been created, the migrate command applies them. Django keeps track of which migrations have been applied already, and only runs the migrations that haven’t yet.
However, if you need to edit the data in the database itself, you have to do a little bit more. There are three main ways to edit the data in the database: running SQL in the database shell, creating and saving model objects in the Django shell, and creating a migration that runs arbitrary Python or SQL.

For one-time changes to the data in the database (e.g. deleting an object that was added accidentally), you can change data with native SQL. If you use PostgreSQL (also known as Postgres) as a database, you can activate the Postgres command line tool with the command `\psql`. After activating the command line tool, you can run any SQL code that you need to view, add, or delete rows. With the `\psql` command you can use SQL to change your database schema, but this is to be avoided. Because any manual changes you make with SQL aren’t recorded in a database migration, Django won't know what's different, which can lead to inconsistencies and unforeseen consequences.

A native Django way to make one-time changes in the database would be to enter the Django shell with the command `python manage.py shell`, access the objects in a model (Django’s way of referring to rows in a table), and delete or change the objects you’re interested in. These methods still end up running SQL in the database, but it’s highly constrained to prevent accidental deletions or overwrites.

The final way to edit the data in a database is by creating a new migration on your model. This migration would either run arbitrary Python or arbitrary SQL on your database. Because each migration only runs once, this method is especially useful for loading initial data, but can also be used to batch edit existing data. This method is the most robust of the three because it is only run once, there’s an easy record of how the changes were made, and can be reversed by writing a function in the migration to undo the effects of the Python or SQL code.

Creating and Using a REST API with Django

Creating - Django Rest Framework

The serialization feature was important for me to avoid slow page loading times. If I load all of the data in the view and send it to the template initially, the client has to wait for all of the data to be pulled out of the database and sent to the client’s browser before the template loads. Unfortunately, this is much longer than most people will tolerate, so I request the data after the page had already loaded. This was a perfect application for making AJAX requests to a REST API, but I needed to make this API.

Luckily, there’s a package that helps with just this: Django Rest Framework (DRF). In their book Two Scoops of Django, authors Audrey and Daniel Roy Greenfeld describe the extensive use of DRF as having caused “DRF has become so ubiquitous that a question is often what is the difference between Django and Django Rest Framework” (Greenfeld and Greenfeld 2017).
Django Rest Framework makes it easy to build browsable APIs and to transform database queries into JSON that can be accessed with RESTful HTTP requests (GET, HEAD, etc.). I used DRF to serialize the data that was processed in different ways. The main section of data that I serialized is what is referred to in the neuroscience community as **raster data**: a matrix that has each time step as a column and each exposure to a stimulus as a row.

I used DRF to translate a matrix of decimals in Python to a list (rows) of lists (columns) of decimals in JSON. Then I specified that it should accept GET HTTP requests and return the serialized data, which set me up to accept AJAX requests from my web app.

**Using - Promises**

When it came time to actually get the data from the database, I ran into a conundrum. Since AJAX requests are evaluated asynchronously, how will I know when I can use the data? Fortunately, there is a built in JavaScript object called a **Promise** that helps deal with exactly this situation.

The Promise object is a representation of the eventual completion or failure of an asynchronous operation and the value resulting from that operation. You specify a Promise by executing a asynchronous operation (including an XMLHttpRequest) and telling the Promise what value it should resolve to. After the asynchronous operation finishes a Promise will resolve to either the specified value or an error. The Promise’s .then() method can then be used to access and operate on the value the Promise resolved to ("Promise" n.d.).

I therefore used Promises for GET requests to get data from the database’s REST API, and used the .then() method to create visualizations only after the data was received. I also used chained Promises to average activation for many stimuli exposures in a less memory intensive way. For every stimuli exposure, I used a new Promise within a .then() call that averages the current average activation and the previous average activation.

**Django and Security**

Django also takes care of many security concerns that can plague web apps. Because information is being sent continually between the client and the server, it’s possible that a client with bad intentions could hijack the communication line and send problematic content through your webapp. Django calls the software that sits between these requests and prevents the site’s misuse **middleware**, and Django has many built in middleware components to prevent specific security threats.

Some specific things that Django protects you from out of the box are **clickjacking**, **SQL injections**, **cross site request forgery (CSRF)**, and **cross site scripting (XSS)** ("Security in Django | Django Documentation | Django" n.d.).
Clickjacking happens when a hacker wraps your site in their site, and thus fools people into clicking on things they didn’t intend to on the malicious site. Django has built-in middleware to avoid this. SQL Injections happen when people try to execute raw SQL on the underlying database of your site. This can lead to hackers deleting some data or the entire database. However, Django’s model querying syntax for accessing the data from the database creates very specific SQL which protects Django projects from such injections. CSRF attacks happen when a hacker utilizes the credentials of a user without that user’s knowledge or consent. Django checks every POST request (like logging in, submitting data, etc.) for a secret specific to each user. This ensures that a malicious user cannot access the POST requests of other users and take their information. XSS attacks occur when hackers store and send pieces of code to other users that are executed on the client side. This usually happens when users store code in a database and that data is rendered on a page.

Though hackers are probably not trying to steal information about neural data, because there’s the possibility that this site might be expanded to accept uploaded neural data, I wanted to start with a framework that was easily scalable rather than have to rewrite everything in a different framework.

Putting it all Together - Serving in Production

To actually connect Django with the HTTP server that could send data-filled HTML, CSS, and JavaScript across the internet, I used a piece of technology called Cookiecutter Django.

Cookiecutter Django is a collection of technologies that make creating a production-ready Django web application easy and fast (Greenfeld 2018).

Cookiecutter is itself a technology that takes a project template and inserts information into the project based on the user’s information that goes into the template. Thus, Cookiecutter Django is a template used to customize a fully productionizable version of Django very easily.

Cookiecutter Django makes a Django web app available at a public URL by either running a Docker container on one of Hampshire’s servers or connecting to a cloud based server on a Platform as a Service (PAAS) like Heroku or Amazon Web Services. I decided to take advantage of the resources that come from attending a college institution and use one of Hampshire’s servers, which I’ll refer to henceforth as “the application server.”

In the Cookiecutter Django template there is a compose file that specifies the collection of services the web application app requires to run and Dockerfiles (one for the entire app and one for each service) that specify environment variables. When you run the `docker-compose build` command, an image of each service (the code, file system, and environment variables in the Dockerfile needed to run the service) is pulled from Docker Hub (an online repository of code libraries and images). This means that an isolated copy of the service is downloaded from
a central repository. For those who are familiar, this is analogous to using pip to install a Python package into a virtual environment. The collection of images that make up the web application can be run with the command `docker-compose up`, at which point they become containers with state (“Overview of Docker Compose” 2018).

An analogy for the relationship between a Docker container and Docker image is that the image is a recipe while the container is the cake. You can make as many cakes as you want from the recipe, and the recipe will never change. In the case of Cookiecutter Django, the cake is a fully functional web application with a large number of features. The image provides instructions while the container is an instantiation of those instructions (“Docker Overview” 2018).

The services that Cookiecutter Django Docker uses are Django (the web framework), Caddy (a web server with built-in encryption from LetsEncrypt), Sentry (a program to log errors in production), Redis (an in-memory data structure store that’s used for caching recently requested data), PostgreSQL (a database), Mailgun (a program to send email from the website), and Celery (a program to run code on the server in an asynchronous task queue).

Once the initial configuration is out of the way (which isn’t too difficult), Docker allows the entire web app to be fully functional in only two commands!

Cookiecutter Django also provides test coverage, the option for live reloading using a Grunt, a JavaScript task runner that waits for changes in the project source code, and immediately refreshes the page instead of having to manually refresh.

I chose Cookiecutter Django in order to be prepared in case my website needs to scale in the future. In retrospect, it took me a long enough time to figure out how to get Cookiecutter Django to work (see the difficulties section), that it may have been a better idea to use a simpler deployment technology, but I learned a ton in the process.

Visualization

D3

For styling existing HTML elements CSS can accomplish a lot for you. CSS does allow for some animation and transition, but does not allow for animations or transitions that respond to scroll or click. This meant that for the interactive web page that I wanted to make, I was going to have to work with a graphics framework that I could manipulate using JavaScript. One of the most popular ways of doing this is by using an Scalable Vector Graphic (SVG). As the Mozilla Web docs state, “SVG is essentially to graphics what HTML is to text” (“SVG: Scalable Vector Graphics” n.d.).
To any HTML element you can add different HTML elements (like headers `<h1>` and paragraphs `<p>`) and group them with a `<div>` element. Likewise, to any SVG element you can add different SVG elements (like circles `<circle>` and lines `<line>`) and group them with a `<g>` element. This means that to create graphics on my web page, I can use JavaScript to specify the collection of shapes that people looking at my web page are going to see.

However, I needed to work with creating visualizations based off of data, and making a simple animation of a SVG scatter plot in native JavaScript would require a lot of work to render each individual point. Luckily, there is a JavaScript library that has been used in a huge number of interactive visualizations, including many of the sites used as inspiration for this project.

**D3.js**, a JavaScript library that stands for Data Driven Documents, more commonly referred to as D3, allows for direct manipulation of elements in the DOM based on data. With D3, you can easily select any HTML or SVG element, assign it some data, then create new elements and stylings based off of that data.

D3 makes animations incredibly easy. You select the element or group of elements you want to make changes to, then call the transition function, then specify the attribute you want to change. D3 will then interpolate the value of the attribute from the current value to the specified value. The attributes that change can be x or y coordinate, color, size, or any available attribute of the selection!

**Graph-Scroll.js**

In order to make my website a true example of scrollytelling, I utilized a JavaScript library called graph-scroll.js that calls a different function for every HTML div element with class “section” as it scrolls into view. Graph-scroll works by adding another class to the div element in view so that it has the classes “section” and “graph-scroll-active” once the div passes 50% mark on the viewport. This new class allows you to change the styling of the div element (like greying out inactive text) in order to make it really obvious when a section is in view.

**Difficulties**

**How to Serve?**

I ran into a number of difficulties while working on this project. One such difficulty was setting up the production server. Because the computer serving neuraldata.net needs to be on all the time, I asked Hampshire college’s webmaster Josiah Erikson for an available computer on which to install Cookiecutter Django.

We first tried installing Cookiecutter Django on the existing Cognitive Science web server located at gibson.hampshire.edu (the computer itself is called Gibson). However, when I did so I could access basic HTML pages and manually import JavaScript, but the data in the database,
template inheritance system, and other capabilities that Django supposedly had weren’t available.

Eventually, we realized that this was due to the fact that port 80 (the port that most websites are accessed at) was already in use. In fact, the caddy server inside the docker container was serving to port 80 of Gibson, but Gibson was already using the open source HTTP server Apache to serve to port 80! Therefore, when I was accessing the web page that should have been served by Django, it was actually being served by Apache. Since Apache doesn’t know how the Django Model-View-Template system works, Apache could only serve the most basic web pages.

I then realized that I needed a server that wasn’t already using port 80 with which to run Django in production. This meant that I needed a different server that could continuously listen for requests for neuraldata.net, and in return the pages filled with data, static images, and JavaScript.

Josiah therefore helped me set up a new server (neuralexploration.hampshire.edu, which I’ll refer to as “the server”) and install Cookiecutter Django on it.

It was then a new challenge to get Cookiecutter Django working on the server. I first thought that I needed to manually install Postgres, Caddy, Redis, and any other technologies that Cookiecutter Django requires to serve a project. Since the server belonged to Hampshire, I was unable to install system wide programs on my own. After a week of requesting a package be installed, getting error messages, requesting another package be installed, and so on, I realized that I had misunderstood how Docker was working.

Docker wasn’t creating an isolated environment on the server that utilized the downloaded programs. Instead, Docker downloads an image of every app specified in the project Dockerfile from Docker Hub when you run the command ‘docker-compose build’ into the docker container. This means that all of the apps Docker needed to run the server are downloaded automatically with one command.

After I figured that out, Josiah and I worked to give me the right permissions to be able to run the docker-compose commands. I then ran docker-compose build to download the apps into the Docker container, and docker-compose up to serve the Django project from inside the container.

Data Size

Another complication that I had to deal with is data size. The dataset that I was initially going to use was a very large MEG dataset. To load the data into the database, I first attempted to use a Django feature called a fixture. A fixture is a file in either JSON or YAML format that can be used to load the data in that file into the database. I wrote a script to convert many MATLAB
data files into one JSON fixture, but when I tried to load the fixture it was too large and caused Django to run out of memory.

I therefore needed another way to load this MEG data into the database. I created a database migration that used arbitrary Python code to iterate through the MATLAB data files and convert and save them as Django model objects. Unfortunately, this process took a prohibitive amount of time - over 18 hours and the data still hadn’t finished loading into the database.

I’m not entirely sure why the data took so long to load. My intuition says that writing data to disk (which is required to save data to the database) may be a bottleneck, because data writing speeds are fixed. Since the MEG data was very large and data writing speed is fixed, it didn’t seem like I could use a simple migration to load the MEG data.

Therefore, I decided to load a smaller dataset in order to work more on the visualizations, and if I had time after that I would figure out how to load larger datasets into the database.

From Tech to Product

Understanding how web pages are organized, computed, and deployed was been a real challenge. After clarifying the technical terms and finding the technology to create my website, it finally became possible to write the code and text for my website available for anyone. Knowing how the internet works and how data is accessed via the internet is incredibly important to make data driven web applications - and visualizing data is one of the best ways we have to understand it.

Visualization Example: Percent Selective Over Time

One of the last graphs that I made on my website was the animated graph shown in Figure 3 that visualizes the p-values calculated from running ANOVAs on each time bin for each neuron. This graph is made using most of the technologies I discussed in the technology section, and thus will be a practical example that serves to bridge my discussion of the landscape of technology that makes up my webpage and the webpage itself.
Figure 3: Visualizing the relationship between p-value and percent of selective neurons over time. 1) Plot of p-values on a negative log scale over time, colored green if they are above the alpha value. 2 and 3) Transition between the p-value plot to the selective neurons plot, the p-values above the alpha value disappear and the percent of selective neurons for each time bin are connected by a line.

When the webpage loads, it automatically sends out Ajax requests to the various REST API endpoints that I set up for different pieces of data. These Ajax requests are stored in Promises. Any computation I want to do with the requested data I simply use the .then() method of Promises that run the function inside the method if and only if the original Promise returned the requested data.

```javascript
my_promise.then(function(data){
    // do computation with `data` here, will only run if `my_promise` has returned
});
```

Figure 4: Example code accessing the result of a JavaScript Promise called `my_promise`
The percent selective over time graph utilizes the Promise called `anova_data` that asynchronously requests data from the ANOVA REST API at the URL `neuraldata.net/spike/anova/(?P<i>[1-3])` where the number after `anova/` is automatically parsed and used as a parameter dictating the bin size.

This parameter is passed into the `anova_from_bin_list` view of the API webpage, and used to query the BinnedData model for the bins of the size specified. The model then translates the query into SQL that it uses to retrieve the binned data from the `visualize_binneddata` table in the database. Upon receiving a row for each neuron from the table (Django calls any model's query results a `queryset`), the view calls the `compute_anova` method on each BinnedData object (one for each neuron) in the queryset. This returns a nested list of the ANOVA values for each time bin and for every neuron, which is transformed into JSON by the Django Rest Framework serializer and then sent back to the client. Figure 4 represents this process visually.

Once the ANOVA data is received by the client, all of the p-values are plotted as thin svg rectangles on a log scale. The fill of the rectangles is dictated by a function that tests whether or not the p-value is less than the alpha level. If the p-value is less than the alpha level, the fill is green; otherwise it's black. The alpha level is represented by a red line, and the value of the alpha level is controlled by the value of a slider that the user can change.

After the p-values are shown, the p-values greater than the alpha levels disappear and the rectangles representing p-values less than the alpha values transition to circles plotted at the percent of neurons that are selective for each time bin. These points are connected by a D3 line with a connecting function that uses a cardinal spline to connect the points in a smooth fashion. Such D3 lines are actually just syntactic sugar that make creating arbitrary geometric shapes with SVG (called an SVG path) easy. The y-axis also transitions from a log scale to a linear one. Every time the user changes the alpha level via the slider, the entire animation is rerun with the new value, resulting in a visual mapping of the relationship between the ANOVA p-value, the alpha value, and selectivity.
Figure 5: How the ANOVA p-values in the graph in Figure 3 are calculated and sent to the client from the server.
Exploring neural data visually
We’ve all wondered how the brain processes information. You may have asked yourself, 'How am I reading this text?' ‘How can I look at an everyday object like a kiwi and tell what it is?’ Neuroscientists ask questions like this everyday, and then conduct experiments to figure out the answer.
This website will explore some common ways that neural data is analyzed in order to figure out how the brain processes information.
Scroll to learn more.

What exactly is brain activity?
Scientists know more about how the brain works than ever before, but there are still many unanswered questions.
To answer these questions, neuroscientists measure brain activity.
The brain is made up of a large number of specialized cells called neurons. Typical neurons consist of a soma (or cell body), dendritic branches (or dendrites) that receives signal from other cells, and an axon that conducts signal away from the cell body to other neurons or to muscles.
When the sum of signals that reach the dendrite from other neurons is larger than some threshold, the neuron "fires" - thus propagating the signal through the neuron’s long axon to reach other neurons.
This propagation of signal is called an action potential, or spike, and is the basis of all neural communication. A neuron can only fire a single action potential, never a half an action potential, so a spike train that measures when action potentials are fired is only ever 0 or 1.
Dendrites constantly receive new incoming signals, and therefore the action potentials that the neuron fires create a pattern over time.
One way that neuroscientists measure brain activity is by implanting electrodes into the brain that record spikes. This collection of action potentials recorded over time is called a spike train and looks something like this:

The experiment
Many neuroscience experiments try to figure out how the brain works by recording responses to different conditions. The experiment we’ll be examining [collected by Ying Zhang in Bob Desimone’s lab in the McGovern Institute at MIT] recorded spikes from 132 neurons in the inferior temporal cortex (a brain region known to be important for visual processing) of a Macaque monkey [1].
On each trial, the monkey was presented with one of seven different stimuli: pictures of a guitar, hand, flower, kiwi, couch, face, or car. Each neuron was recorded for 1000 ms, and the stimulus was shown 500 ms after the monkey fixated a spot at the center of a display. Each image was presented around 60 times while the neuron spikes were recorded.

Averaging over time
Now that we have this collection of zeros and ones, we can ask more interesting questions. Namely, what does seeing a kiwi look like in the brain? At what time can our brains tell that we’re looking at a hand? How can we be sure that these zeros and ones aren’t just due to chance?
To present this data a different way, we can look at the average firing rate over an interval of time within each trial.
This makes it easier to see how neurons’ firing rates differ from each other.
Some neurons fire all the time, some neurons rarely fire, but how can we tell which neurons are important for our experiment?

Averaging over trials
The first thing we want to do is look at multiple trials.
The spike train from one neuron can fire very differently from trial to trial, even when both recordings come from the same stimulus. It's therefore important to look at many recordings from the same neuron to make sure you aren't confusing random activity for activity driven by the stimuli.

Accounting for variability
If we had data from an infinite number of trials, we would be able to calculate the true average firing rate for each neuron and stimulus perfectly. However, because we only have data from around 60 trials for each stimulus, our estimate of the true average firing rate is most likely slightly off. The colored bands that you see are called 95% confidence intervals, and they are created such that the true average firing rate will be contained in the band 95% of the time.
If we have less data, then our confidence intervals have to be larger to capture the true average firing rate 95% of the time. Below you can move the slider to see the mean and confidence intervals change when you use less data. For more information about confidence intervals, see [2].

Looking at different neurons
Now that we have a good idea of what the firing rate for one neuron looks like for every stimulus over many different trials, we can now look at the average of many different neurons.
Comparing neurons

Every neuron responds to stimuli differently. Some neurons naturally fire very often and have a high average firing rate, like neuron A, but other neurons naturally have a much lower average firing rate, like neuron B. Therefore, to compare these neuron firing rates more effectively, we put them on the same scale - a process called normalization. To do this, we calculate the mean and standard deviation of all time bins and trials for each neuron. We then normalize the neuron's activity by subtracting this mean and dividing by this standard deviation. Now instead of plotting average firing rate over time, we plot the deviations away from the average firing rate over time.

Aggregating across neurons

Before when we combined information from multiple trials, we only averaged together the trials when the same stimulus was shown. This allowed us to be able to tell whether or not a neuron responds differently to different stimuli. Similarly, averaging together all neurons would make it difficult for us to tell if the neurons respond differently to different stimuli. For example, neuron A spikes a lot after the guitar and flower stimuli appear, while neuron B spikes more for the couch stimulus but overall less frequently after every stimulus appears. However, if we average the two neurons, then we can no longer distinguish the couch, guitar, or flower stimuli as easily!

We therefore need another way of combining information from different neurons that helps us figure out the answer to our experimental question: how does the brain process and store information about different images? Since we can’t as easily figure out if our experimental condition had an effect from averaging our entire neuron population, we need another technique. As we saw before, the same neuron can respond very differently to multiple exposures to the same stimulus. Thus we will use a statistical method that compares the distribution of average firing rates for each neuron and across every stimulus. We can then run an ANalysis Of VAriance (ANOVA), a procedure that helps us figure out if the neurons actually respond differently to different stimuli or whether the recorded responses are random fluctuations. More technically, ANOVA computes a statistic that gives us a measure of the likelihood that splitting the trials randomly could explain more of the variation in all of the trials than splitting them by stimuli.

We run a separate ANOVA for each time bin and neuron, and get a p-value that gives us the probability of calculating a statistic more extreme than the one observed if the mean firing rate responses were in fact the same for each stimulus. In this analysis, we’re going to set a threshold (called an alpha-level) where we call a neuron "selective" if it has a p-value less than that alpha-level. A very small p-value indicates that the neuron responded differently to different stimuli.
We expect that before the stimulus was shown there is a fraction of neurons equal to the alpha-level that would be called selective by chance. If the fraction of selective neurons is much higher than the alpha-level after the stimulus was shown, that means that the stimuli statistically produced distinctly different neural responses in our experiment. In our ANOVA analysis, we analysed whether the population of neurons was selective by running an ANOVA analysis separately on each neuron. Other approaches, such as neural population decoding, apply a multivariate analysis to look at the population as a whole rather than analyzing each neuron separately [3].

In conclusion
The brain is complicated, and neuroscientists are still trying to answer basic questions about how the brain works. However, the data analysis techniques explored here can help us understand how long it takes information to propagate throughout the brain, how neurons respond more to certain stimuli than others, and about the proportion of neurons that statistically respond to the experimental conditions. As we continue to collect more data and build better data analysis methods, the more we will improve our understanding of how the brain produces complex behavior.

Wrapping Things Up
The web is an amazing medium for learning and communicating. Visual, interactive web pages can help people understand concepts they’ve never seen before, integrate data and graphics more tightly into a cohesive narrative, and make concepts more accessible to visual learners and those who couldn’t otherwise afford textbooks. My Division III project set out to tell a story about neural data analysis. The story that I ended up telling was a story about variability and similarity - how levels of abstraction and aggregation can produce surprisingly consistent insight from seemingly volatile data.

I hope to build on this website more in the future, expanding its scope to multivariate analyses and a discussion of information theory. The beautiful thing about the web is that it isn’t space constrained and thus https://neuraldata.net has the potential to be improved and lengthened as much as I see fit. Furthermore, since the code for the site is on GitHub, anyone else who wants to improve on the site and add more information can put in a pull request for the feature they want to add and it can be running on the server in a few minutes.

Both neural data analysis and web development are complicated topics, but now at least one of them has an interactive web-based visualization that explains how it works.


MDN Web Docs.  