Data Mining Human Reasoning: Vaccine Hesitancy in the USA

Taha A. Kashaf
ID: 180020889
AC40001 Honours Project
BSc (Hons) Computing Science
University of Dundee, 2022
Supervisor: Prof. J. Lawrence

The purpose of this project was to use Machine-Learning powered Sentiment Analysis and Natural Language Processing techniques to classify sentiment about vaccination from textual data in the form of tweets.

The project endeavoured to present the results of this analysis in a clear, easy-to-approach way. Allowing users/readers to be able to view the results alongside other statistical data to derive correlations and comparisons for themselves.

The goal being to facilitate an understanding of the potential underlying reasons for vaccine hesitancy, to be able to better address it in the future.

While many parts of the project were successful – data collections, formatting, cleaning, pre-processing and presenting. Ultimately, the Machine-Learning powered Sentiment-Analysis model wasn't accurate enough to draw useful conclusions from – however, this was primarily due to the lack of training data, and the model can be improved with simply more training.

1 Introduction

Vaccinations and vaccines have become a controversial talking point in this day and age. In the midst of the worlds best-documented pandemic: COVID-19. Many of us saw the development of COVID-19 vaccines as a fantastic feat of collaboration between the global medical and scientific communities. We believed that with the help of the vaccines and large-scale inoculation, multiple years of lockdowns, restrictions and sacrifices would come to an end.

However, while that may be the predominant opinion, many people also look at vaccinations and vaccines as a bad thing. Hence the talking point of vaccines being one that brings controversy to the table.

Anti-vaccination rhetoric has been around since long before COVID-19[1]. However, in recent years, and especially with the advent of social media, the visibility of the movement has grown dramatically.

In pre-COVID times, anti-vaccination rhetoric was primarily something we'd hear about and shake our heads at. For most, it did not have any tangible real-life effect, it wasn't likely to cause any changes to our day-to-day lives.

However, currently, in the midst of a pandemic. With reports of hospital urgent care wards being filled with primarily unvaccinated covid patients[2] the effects of the anti-vax movement are bigger and closer than ever. Where even a vaccinated individual may be unable to receive care due to hospital beds being occupied by those who choose not to take vaccines.

The problem is an obvious one: large groups of people carry a negative sentiment towards vaccinations. The goal of this project was simple: To use Machine-Learning powered Sentiment Analysis, in combination with Natural Language Processing techniques to aid in producing a data-driven solution that would help us understand *why* people have the opinions they do regarding vaccines.

This is obviously not a solution to vaccine hesitancy in itself. With a topic as complex as human reasoning, and studying the choices people make people. First, you need a comprehensive understanding of *why*. Only then, once that understanding has been built, can you begin to tackle the problem itself. This project is an attempt at understanding the reasoning behind the problem.

2 Background

My background research was split into three distinct categories, Data Analysis, Sentiment Analysis Techniques and Previous Attempts at Classifying Vaccine Sentiment on social media.

2.1 Data Analysis

The field of data analysis is a large and still rapidly expanding one. One that has also come to the forefront of discussion in the last few years, as the power of data-analysis and data-driven solutions has been made clear to the general public – often in a negative light.

Perhaps the most famous example of how data analysis has changed the world is the role it played in recent large-scale political campaigns[3]. These successes have even led to legal debate regarding data privacy, and the uses of data analysis. The most well-known result of such debate being the passing of legislation such as the General Data Protection Regulation (GDRP).

As we can see, there is no question about the importance or research potential within big data. The ability to analyse and extract value or meaning from large-scale unsorted data is an ever-growing field with proven real-world results.

2.2 Sentiment Analysis Techniques

For the purposes of this project, I made use of Sentiment Analysis, a form of data analysis on written text. Sentiment Analysis comes in different forms[4], primarily one of two: Lexicon based approaches and Machine-Learning based approaches. Machine-Learning based approaches are then also split further into supervised, unsupervised, and semi-supervised.

Lexicon based sentiment analysis approaches are techniques that make used of a predefined list of words that have been assigned a specific association. These predefined sentiment lexicons are then used to assign a polarity value to each text document (for example, a tweet) by following a basic algorithm[5]. Predefined sentiment lexicons are often referred to as corpora or dictionaries.

Examples of popular lexicon-based sentiment analysis models include the "Valence Aware Dictionary and Sentiment Reasoner", known simply as VADER[6]. VADER is even specifically tuned to sentiments expressed in social media, however, was not suitable for the project, as discussed in *Implementation*.

Machine-Learning based sentiment analysis techniques do not use any form of pre-defined corpora. Instead, in the case of supervised machine-learning, models are trained using full text documents (tweets) that have been hand labelled into certain categories. In the case of sentiment analysis, these are often positive/negative/neutral. This was the approach I decided to take myself and is again discussed further in *Implementation.*

Semi-supervised, or hybrid approaches to sentiment analysis use a combination of lexicon, and machine-learning based techniques.

2.3 Previous Attempts at Classifying Vaccine Sentiment

My project is not the first attempt at classifying vaccine sentiment, it is not even the first attempt at classifying vaccine sentiment on twitter specifically. In recent times, similar studies have been carried out[4][7][8]. However, these studies primarily focus on classifying data and presenting the classification results in a purely academic format, such as this report. While this provides an excellent base for that classification data to be used or extended for further study, this was not the direction I wanted to take my project in.

These previous projects and studies typically used a Semi-Supervised Machine-Learning methodology. Meaning a combination of Lexicon-Based and Supervised Machine-Learning techniques. This approach, combined with a high amount of training data lead to a high level of accuracy. In the case of one study, with 10,500 points of training data, a >85% accuracy was achieved in classification[4].

However, even with multiple studies with high levels of classification accuracy, my primary takeaway was that none of the information gathered was presented in a wide and easily accessible scope. It remained within the realms of academia, as mentioned earlier. This is where I wanted my project to be different, deciding from early on that I would present my results in an easily accessible, graphical format for anyone to be able to find and use. I discuss this further in the following *Specification* section.

3 Specification

I split my project into four broad, high-level sections early on:

1) Data Collection
2) Data Formatting
3) Data Classifying
4) Data Presenting

Each of these sections also had distinct sub-sections, some of which I was aware of at the outset of the project, others I discovered as the project went on.

Due to this, the nature of the project was very exploratory for me. I was learning what was required as I was progressing through the stages. This made it very difficult to nail down a development methodology early on in the project. I discuss this further in subsection *Development Methodology*.

So, while parts of my specification were pre-planned right from the outset, as the project evolved and I learned more, the initial plans rapidly fell to the wayside. This led to me spending less time in the pre-planning and specification writing stages, and more time just working. As time spent

planning was often time wasted due to how quickly reality deviated from plans.

3.1 Data Collection

The first part of any Data-Driven solution is to acquire the data. For this project, I would be using the Twitter API[9] for my data collection purposes. The project plan for this section broke down as the following:

1) Acquire Twitter Developer Account
2) Create Twitter API Application (needed for endpoints access)
3) Learn how to use the API 2.0 Endpoints
4) Use API to Collect Data.

However, as stated earlier in *Specification*, reality rarely went to the original specification, even at such a broad high-level breakdown. So, the Data Collection part of the project went more like the following:

1) Acquire Twitter Developer Account
2) Create Twitter API Application (needed for endpoints access)
3) Escape from Twitter Spam Filter detection
4) Learn how to use API 2.0 Endpoints
5) Collect Data
6) Learn how to use API 1.1 Endpoints
7) Collect Data

The details as to why reality deviated from the original specification so dramatically are discussed thoroughly in *Implementation.*

3.2 Data Formatting

The data formatting section was planned as the following, with no changes drastic during implementation:

1) Data "Cleaning"
    a. Discarding extra information returned by the API that was unnecessary for the project goals.
2) Data Structure "Formatting"
    a. Splitting the tweets into individual text files, organizing them into a two-level folder structure that the Machine Learning model could use.
    b. This stage was done in conjunction with hand-labeling data for training
3) Data "Pre-Processing"[10]
    a. Remove all special characters
    b. Remove all single characters
    c. Remove single characters from start
    d. Replace multiple spaces with single spaces
    e. Removing pre-fixed "b" from loading function
    f. Converting to lowercase
    g. Lemmatization
    h. Covert Text to Numbers
        i. Done using "Bad of Words" Model.
    i. Removing Stopwords
    j. Finding TFIDF – (Term Frequency – Inverse Document Frequency)

3.3 Data Classifying

Once all the data has been adequately formatted and pre-processed, the final classifying stage is fairly simple. Only consisting of the following:

1) Train Model
2) Evaluate Model
3) Save ("Pickle") Model
4) Use Model to Classify remaining large-scale data

A few snags were hit in this area and are discussed in detail in *Evaluation / Testing*.

3.4 Data Presenting

The final part of the project involved Data Presenting. This was making a full front-end website that users could browse to see the results of the project. As well as to see some comparisons of sentiment data alongside other statistics. I arrived at the decision to implement this part of the project based on background research. This was what would set my project apart from previous studies into Vaccine Hesitancy – a clear, non-academic, easily accessible way to view the classified data from the project.

This was further motivated by my first meeting with my project supervisor – Prof. John Lawrence, where we discussed how the project would be split on a back-end/front-end level *(see Appendix A)*.

This would also be used as an opportunity for me to learn and become more familiar with JavaScript, and JavaScript libraries such as React.js[11]. While I had done front end development before, it had not been with simpler technologies, so this was a chance expand my skillset. As a result, I arrived at the following (loose) plan/specification for the website.

1) Learn React.js
2) Create landing page
3) Create "Comparisons" Pages
4) Present data graphically using Graph.js[12]
5) Achieve full deployment for website.

3.5 Development Methodology

While *Development Methodology* is not explicitly related to *Specification* it is important in my case to understand the development methodology I implemented, as it was directly

responsible for the very loose style of specification I arrived at and gave myself for the purposes of this project.

The initial plan for the project, before I ever had my first meeting or read my project brief, was to adopt an Agile Methodology to development. I had in previous projects adopted Agile practices to great success and was a fan of how Agile allowed flexibility and rapid prototyping during development.

However, after my first meeting with my project supervisor, it became apparent to me that I knew very little about the space in which my project was going to be developed. I had never worked with the Twitter API, with Machine-Learning, with Sentiment Analysis or with Natural Language Processing techniques, even my plan to use React.js was so that I could have the opportunity to learn it. Every single stage of my research and development process would be like uncovering fog-of-war on a map – I could only see what's in front of me as I came up to it.

As a result, although I did initially plan sprints *(see Appendix B)* I quickly decided it wasn't a good investment of time to generate detailed requirements or user stories, as these requirements would quickly be overridden during development – when I actually learned what is required, rather than guessing without much knowledge.

4 Design

My design choices were across a number of areas, what languages to adopt, what frameworks to use, what tools would I be able to use, what constraints would I be working with and what systems would I be using for backup and version control.

Each of these decisions was considered for each of the main areas detailed earlier in *Specification* (Data Collecting, Formatting, Classifying, and Presenting).

However, before any of that could be considered, the first and simplest decision needed to be made regarding backup, code storage, and version control.

4.1 Backup, Code Storage & Version Control

This was the easiest design decision out of them all. While consideration was given to services such as SourceForge[13], BitBucket[14], GitLab[15] and even just locally saving all my work. The final decision was very quickly to use the industry standard: GitHub[16].

Although I had been recommended the alternatives by my friends, and even initially wanted to just work locally from my own machine. I had prior experience with GitHub and decided that even though the project was a solo one, and did not require any code-collaboration, I would use GitHub just because it provides a form of backup – just in case something was to go wrong. (To good effect too, as I did lose all my local work during the year due to a system malfunction on my main machine that required a full re-install)

4.2 Data Collection

For Data Collection, right from the outset I had decided to use the Twitter API. This wasn't much of a choice, although techniques such as JavaScript Web Scrapping exist for data collection, the Twitter API offers the most control and easiest level of access to Twitter Data.

Within the Twitter API, a decision must also be made between using the version 2.0 Endpoints, and the version 1.1 Endpoints.

Initially, I decided on using the 2.0 Endpoints, as the version 1.1 Endpoints were both deprecated, poorly documented and no longer maintained.

To use the 2.0 Endpoints, as recommended by the Documentation I used the Postman API Platform[17] tool. The Postman API Platform is a tool for developers to design, build, test and iterate their APIs. It allows user to make requests to APIs from a graphical user interface. This was perfect for me, as I had never used the Twitter API before and a GUI-centric way of accessing it would be the simplest.

Twitter also supplied a pre-made collection[18] that made access to the Twitter API 2.0 Endpoints from Postman very simple. While this worked, due to reasons discussed in detail in *Implementation* I was forced away from using the Twitter API 2.0 Endpoints and had to pay for access to the 1.1 Endpoints, and use those.

The version 1.1 Endpoints provided by Twitter are both deprecated, and poorly documented. This meant there was no Postman collection available to allow the API to be accessed graphically. This lead to a steep learning curve as I was forced to adopt Ubuntu (via Windows Subsystem for Linux – WSL[19]) and cURL[20] (a command line tool for transferring data) as my main technologies for Data Collection.

4.3 Data Formatting

The next design choice to be made was what technologies would I use for the expansive amount of data formatting that was required – in fact, this was probably one of the most key parts of the project. As such, the design choices made here would be consequential.

Luckily, these choices were somewhat made for me. During my first meeting with my project supervisor. I was strongly advised to use Scikit-Learn[21] for when I eventually moved

on to the Machine-Learning and classifying stages. Scikit-Learn is a Python[22] library. On top of this, my own background research showed that Python was considered the go-to language for entry-level Machine-Learning and Data Analysis. So, Python seemed like the obvious choice, if I used it now during the Data Formatting stages, I would be more familiar with it during the Data Classifying stages.

However, although Python seemed like the obvious choice, I was also going to be using JavaScript later during the Data Presenting stage, when building my front-end. The Data returned from the Twitter API 1.1 Endpoints was also returned as JSON files – JavaScript Object Notation *(see Appendix C)*. So, I was initially tempted to use JavaScript for all the Data Formatting purposes.

Despite this, my final decision was to use Python. As I learned that the Data Formatting and Data Classifying stages would be heavily intertwined and switching languages between them would just be extra work for no benefit.

Within Python, a number of packages were used to aid in the Data Formatting. These include:

- NumPy[23], a library that adds support for large, multi-dimensional matrices and arrays, as well as a large collection of high-level mathematical functions to operate on those arrays.
- re[24], a base python module that provides regular expression matching operations
- Natural Language Toolkit (NLTK)[25], a suite of libraries and programs for natural language processing for English.
- pickle[26], a base python module for serializing and de-serializing Python object structure – essentially granting the ability to "save" trained Machine-Learning models to a file.

4.4 Data Classifying

As stated in the previous subsection *Data Formatting*, the advice given during the initial project meeting *(see Appendix A)* was to use Scikit-Learn, and by extension, Python, for the Machine-Learning aspects of the project. My own research also showed that Python was by far and away the most popular language for entry-level Machine-Learning.

However, during my research, I also discovered alternatives to Scikit-Learn. Primarily in the form of TensorFlow[27], TensorFlow being a free and open-source library developed by Google that is available in Python. It can be used for a wide range of tasks but has particular focus on training and inference of deep neural networks and machine-learning.

TensorFlow also had a Sentiment-Analysis tutorial available in their own documentation, whilst Scikit-Learn did not. However, upon discussing with my project supervisor and being provided a number of resources *(see Appendix D)* for Sentiment-Analysis using Scikit-Learn, I did finally settle on that.

For the purposes of the model itself, I settled on using a Random Forest Classifier algorithm. While the Random Forest Classifier algorithm creates models that are slow to train and require higher processing power. This wasn't a large problem on my end, as the model would only need to be trained once, before being pickled. Random Forest Classifiers are also good at working on large datasets, my project was designed to be easily extendable, so I would need a model that could cater to a growing dataset, which the Random Forest Classifier was suited for. The following reasons, as well as Random Forest Classifiers being recommended by various tutorials and articles, led to that algorithm being the one I very quickly settled on for training and classification purposes.

4.5 Data Presenting

For the Data Presenting portion of my project, I was certain from the start that I would be making a website as my front end. I also knew that I wanted to use this as an opportunity to learn and work in JavaScript. As a result, although a number of JavaScript frameworks such as Angular.js[28] and Vue.js[29] are considered "better" or "easier"[30], I decided on using React.js[11].

This was because regardless of ease of use, or being told other frameworks were "better", React.js is by far the most popular web framework in use today[31]. Which means it is a valuable technology to be proficient in as a developer.

As a result, it didn't get given much further considering, I would be using React.js for my website/front-end. This was not the only technology used however, several packages were used alongside React.js to develop the final website, including:

1) Bootstrap[32], a free and open-source CSS framework for responsive front-end web development
2) Chart.js[12], a free and open-source JavaScript library for data visualization
3) React-Router[33], a library for routing in React.
4) React-Twitter-Embed[34], a library for easily embedding tweets, allowing you to show tweets without violating Twitters style guide/policy.
5) React GitHub Pages[35], scripts that allow deployment of React Apps to GitHub Pages. Allowing a fully-deployed website.
6) React-Select[36], a flexible input select control library for React.js

5 Implementation

Since the project is easily divided into the four main sections seen previously in *Specification* and *Design*, the *Implementation* section will follow the same structure, starting with Data Collection.

5.1 Data Collection

While this should have been a very simple part of the project, due to the clear idea of the type of data I wanted to be collecting, it ended up being the section that took the largest period of time, both because of the learning curve associated with using the Twitter API for the first time, and also with factors that occurred outside of my own control with the API, and with my API access.

5.1.1 The Trouble with Twitter

Whilst the Twitter API puts a large sea of data right at the fingertips of developers, accessing this data is a challenge beyond just the technical aspects of interacting with the API.

I first made my application for a Twitter Developer account in March of 2020, well before my 4th year or before this project. I include this piece of information as that application was not approved until over 6 months later. This should give a clue about how quickly things move at Twitter.

This became a severe issue during implementation when I received the following email seen below, in *figure 1*.

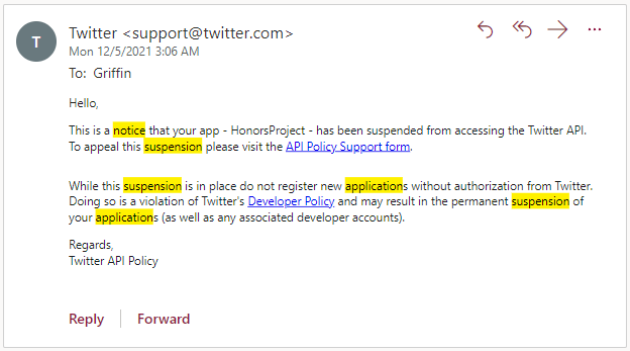

Figure 1: Email from Twitter informing that my API Access had been suspended.

This email began an approximately 6-week period in which little to no development work was able to be done, as I had been locked out of being able to complete step one for a data driven project: collect my data.

I read the developer policy, and the API Policy back-to-front to make sure I wasn't in violation of any rules. Which fortunately I wasn't. However, I already knew this as my API access was suspended without hours of creating an application and receiving API keys – before I'd ever even sent one request.

With little else to do, I filled out an Application Appeal, and waited. After a few days had gone by, I had received no response, not even an acknowledgement of my appeal being received. At this point I deleted the Application that had been suspended and tried making another one – just in case I had done something wrong in the creation process.

This changed nothing, and once again, within hours of creating the Application and receiving API Keys, before ever sending a request, I received another email identical to the first. Detailing that my Application had been suspended from accessing the API. Once again, I filled out the Appeal, once again I received no acknowledgement of any kind from Twitter.

At this point, knowing how long it took my Developer Account to be approved in the first place, I decided to take matters into my own hands (with what little I could do), and search for other people who had similar issues with Twitter, hoping someone had found a solution or explanation.

After weeks of deep diving the Twitter Developer Forums, which were filled with complaints of the same nature, but no solutions, I found one random comment in one random thread explaining that Twitter automatically suspends all applications they expect are coming from "bot" accounts, and does not consider any appeals for these applications, or these accounts.

At this point, I put two-and-two together. Realizing that my Twitter Developer Account was linked to my normal Twitter Social Media account (as all Developer Accounts need to be). My normal account had no activity on it, as I do not use Twitter. No tweets, no comments, no followers, no activity of any kind. So, as a result, I was being considered a "bot" or "ghost" account, and having my applications suspended and my appeals ignored.

The solution for this, quite painfully, was for me to resurrect long since dead and inactive social media accounts, so that I could beg people I know to follow me. All in the hopes that my Twitter Account would meet the bare minimum threshold required to not be considered spam. These embarrassing messages are captured in *Figures 2* and *3*.

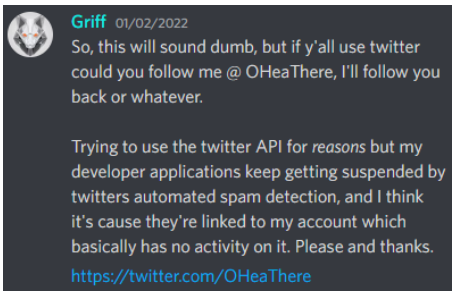

Figure 2: I beg for followers on Discord.

Figure 3: I beg for followers on Facebook

After these messages, I waited patiently for a number of days, to allow people to see them and follow me before I attempted again to submit an Appeal. When I did finally submit the Appeal to have my API Access restored, I did this time receive an acknowledgement that my appeal had been received, not an unsuspension, merely an acknowledgement *(see Appendix E)*.

Even after that, it took a further eight days, up to the 12 of January for my Application to finally be unsuspended, and for my API access to be returned *(see Appendix F)*. Overall, this ordeal during the *Implementation* stage cost me approximately 6 weeks of development time, from early December to almost mid-January.

5.1.2 Twitter API v2.0 Endpoints

After receiving API access back, it was time to rapidly move on and begin collecting Data. I had settled on a plan to collect Tweets from each state in the USA that contained one of the following key words: “Vaccination”, “Vaccine”, “Vaccinated”, “Vaxed” or “Vaxxed”.

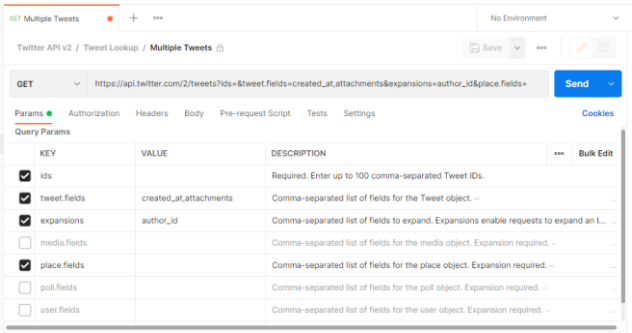

Figure 4: The GUI Presented by Postman to simplify making requests to the API

Whilst I was successful in using Postman (as seen in *figure 4*) to make requests to the API and to retrieve data, I hit a snag that unfortunately I was unable to solve itself. The API access that I had been given was akin to a “Student” Level account, elevated from normal API access, but still without all the bells and whistles available to me. This led to two issues. The second of which I would only become aware of further down the line.

The first issue, and one that was initially very worrying was that I did not have access to any form of Geographical Data. I was able to query Tweets that contained specific words, like vaccine, but I was unable to query for those tweets to also be limited to a certain State – like New York. This put a massive hole in my plans, as the entire idea behind the project was to present comparisons of Vaccine Hesitancy alongside other statistics, which I had planned on doing by comparing and contrasting different American states.

While this first issue was never resolved itself, as that would require “Researcher” level access to the API (which I did apply for, and still have not received any sort of response), it could be circumvented. This circumvention/solution was provided by the project supervisor, Prof. John Lawrence, during one of our meetings. Whilst I was unable to restrict tweet sources geographically, I could still restrict them based on what was IN the tweet. So, the solution because to simply search for tweets that mentioned both one of the vaccine related key words, and the name of the state I was collecting data for.


```
{
    "data": [
        {
            "created_at": "2022-02-12T12:25:00.000Z",
            "id": "1492474818507128833",
            "text": "I'm telling you, with just one or two buddies, I
            could go on the internet and convince the MAGAs that the
            entire water supply of Arkansas was secretly tainted with
            covid vaccine and they'd literally start smashing their
            kitchen sinks in protest"
        },
        {
            "created_at": "2022-02-11T03:36:29.000Z",
            "id": "1491979426942308352",
            "text": "In just a very few days Arkansas will tally its 10,
            000th Covid death.\n\nIt has been 10 months since the vaccine
            was opened up to all Arkansans 16+. \n\nIn those 10 months,
            we've logged about 44% of our total Covid deaths. The vast,
            vast majority of them completely preventable."
        }
    ],
    "meta": {
        "newest_id": "1492474818507128833",
        "oldest_id": "1491979426942308352",
        "result_count": 2
    }
}
```

Figure 5: 7-Day Results for Arkansas provided by the v2.0 Endpoint

This was when I was presented with the second of my two issues with the API v2.0 Endpoints. Once again, due to my limited access level to the Twitter API, I was unable to access the “full-archive” search option that was provided, instead being limited to only able to query the “7-Day Archive”. I didn’t initially think this was a huge problem, as my goal was only to collect 500 tweets from each state, for a total of 25,000 data points. I assumed far more than 500 tweets about vaccines (containing state names) would

be made in such a time period. However, I was very wrong. The query that returned the most results was for the District of Columbia, and even that contained only a measly 98 tweets. For most states, the results contained less than 10 Tweets, and for some, no tweets were found at all. *Figure 5* shows the results obtained for Arkansas – a grand total of 2 Tweets. The full data collected using this method can also be found in *Appendix G*.

At this point I realized that the v2.0 Endpoints would not be able to provide what was necessary for this project. I would have to collect my data using either external sources (not the Twitter API), or figure out the v1.1 Endpoints, which were both deprecated and poorly documented.

5.1.3 Collecting Data Externally

Before attempting to figure out the v1.1 Endpoints, I decided to look online to see if an existing data set was easily accessible for my purposes. I didn't want to use a Data set that I didn't collect myself, as a key part of this project for me was doing everything from scratch – and being unable to complete the data collection step and simply using existing data sets from the Internet was not part of my plan.

However, I needed to swallow my pride and have a backup ready before I invested time into the v1.1 Endpoints, as at this point I did not know if that would work either, and if it didn't I would be left at square one, with no data, no project, and half the time I had spent simply fighting Twitter with nothing to show for it.

Luckily, I was able to find a passable dataset[37] on Kaggle[38], an online community of Data Scientists and Machine Learning practitioners (Kaggle also happened to be a subsidiary of Google).

This dataset wasn't ideal, as it contained tweets from all over the world, some geo tagged, some not geo tagged, so I was unsure if I'd be able to filter out enough tweets from each individual state to suit my purposes. However, it did contain over 350,000 tweets (*see Appendix H*), so I was satisfied with having it as a last-minute fall-back, should my efforts with the v1.1 Endpoints fail.

5.1.4 Twitter API v1.1 Endpoints

The data that I finally did acquire and use was from the Twitter API v1.1 Endpoints. As has been mentioned a few times, these endpoints are no longer maintained, and are not well documented.

A further issue during implementation with the v1.1 Endpoints was due to their deprecated nature – no existing collection existed for use with Postman. The API could only be accessed from the command line. However, this was not an insurmountable problem, merely a learning curve with regards to using cURL[20].

The issues that were presented with the v1.1 Endpoints were much like the issues with the v2.0 Endpoints. Primarily – I still could not access any form of geographical data, and still could not access the Full-Archive search feature. The search time was however increased from 7 days to 30 days.

For the geographical data, I was planning on using the same workaround as previously, however, as I still could not access the Full-Archive search, the ability to collect enough data was still a worry.

Luckily, after spending time learning the v1.1 Endpoints, I became aware of a potential solution that was not present with the v2.0 Endpoints. For the v2.0 Endpoints, the only way to access to the higher-level functions was to be approved for a Research or Enterprise level account, neither of which I qualified for (although I did apply for a Research account).

However, the v1.1 Endpoints offered the ability to purchase premium access, at quite a steep price. I was able to purchase full-archive search permissions for $100 USD/Month *(see Appendix I)*. This was both expensive, time-limited to one month, and still only offered 100 total requests to the API. So, any bad requests, errors, or other faults would be slowly chipping away at the limited resources I had available.

After all of what I've described, the v1.1 Endpoints and paid premium access was what I ended up using, after spending a large amount of time searching through outdated documentation, the cURL requests that I needed for my purposes were figured out, and example of which can be seen below, in *figure 6*. The API Access token have of-course been redacted.

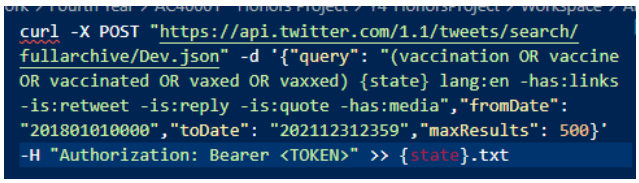

Figure 6: cURL Script Used to Retrieve Tweet Data.

The script includes '"maxResults": 500', as this was the maximum amount of tweet objects that the API allowed to be returned in one request.

Due to only having 100 requests total, and some of them being wasted whilst I was learning the correct syntax, I could only allocate one request per state – which led to 500 tweets per state. Or 25,000 total tweets collected. These tweets were returned in large JSON files – one for each state. An example of which can be found in *Appendix C*.

All the tweet data I received was stored on GitHub – alongside everything else in my project. This led to an

ethical concern, as the data returned by the v1.1 Endpoint differed to the v2.0 Endpoint in that each Tweet was returned as a full Tweet Object. Meaning, all the information about the Tweet Author, and their account was also being returned. An example of just some of the extra data returned can be seen below in *figure 7*.

Figure 7: An example of the extra user data returned with each tweet

I was worried about all the extra user data that was being returned and stored publicly in the project repository (*see Appendix J*). I raised this point during a meeting with my project supervisor, and the discussion agreed to the point that all the data being saved was also publicly available on the given Twitter accounts or Tweets. So, as long as I was not violating Twitter Policy, I was not ethically in any danger – as the data being saved is actually public data, not private.

5.2 Data Formatting

Now that I had all my data, and I was satisfied that it was being ethically stored. I was able to move onto the second stage of my project – Data Formatting, making sure that the data I had collected could be used by my Machine-Learning model.

As discussed in the *Design* section, this fell into three subcategories. Which I dubbed Data "Cleaning", this was parsing the raw JSON files returned by the API, and only keeping the few fields that would be useful, such as tweet text and ID, compared to the almost 100 fields that were actually returned for each tweet.

The second step was what I called Data "Formatting", which is a very broad term, and indeed one that encompasses this entire section. However, in this case, I used it to define the actions taken to split the tweets into individual text files and arrange them in a two-level folder substructure. This was required as the Machine-Learning algorithm used this format, with the folder names being used as the supervised signal labels that the model would be trained with.

This also meant that the step of hand-labeling a certain amount of the data to be used to train the model had to be done at this stage. Since the folder names were being used as the supervised signal labels (positive/negative/neutral). Thus files placed into these folders needed to be classified as well.

The third step was the Data "Pre-Processing", this was finally taking the text data and performing a certain number of operations on it to prepare the data before it was fed to the Machine-Learning Model. These individual operations have already been named in subsection *3.2 Data Formatting* under section *3 Specification*. However, they will be discussed in more detail later in this section.

5.2.1 Data "Cleaning"

Taking the files returned from the API and discarding the extra fields returned for each tweet object was a simple enough task.

This was handled with a quick Python script that can be found in *Appendix K*. The script very simply picked out the tweet ID field, and either the full tweet text field, or the

tweet text field (depending on the length of the tweet). It took the data from these fields and simply printed it to a .txt file for clearer access for the rest of the project. An example of the resulting .txt file can also be found in *Appendix K*.

5.2.2 Data Structure "Formatting"

After the tweets had been "cleaned" and placed into a .txt file for each state. They still needed to be split into individual text files, one per tweet. This was necessary as the Machine-Learning model required the data to be structured in this way.

Apart from being in individual text files, the data also needed to be organized in a two-level folder structure, using the folder names as supervised signal label names. An illustration of this can be found below, in *Figure 8*.

Individual samples are assumed to be files stored a two levels folder structure such as the following:

container_folder/
  category_1_folder/
    file_1.txt file_2.txt ... file_42.txt

  category_2_folder/
    file_43.txt file_44.txt ...

The folder names are used as supervised signal label names. The individual file names are not important.

Figure 8: Explanation of folder structure, from Scikit-Learns "load_data()" function[39]

Due to the files not only needing to be split, but to be organized into the correct categories, I also took this as the opportunity to hand-label 100 tweets from each state (so 5000 total tweets) that I would then use as my training data.

This was once again done using a simple Python script. It would read the data from the .txt files created by the script described in the previous section and present the first 100, line by line in the default output to be classified, before creating and placing them in the right folder/file based on the user classification – given as a simple one letter input to the console.

The script used for this can be found in *Appendix L*, and the resulting generated "training sets" structured as described in *Figure 8* can be found in *Appendix M*.

The act of hand-classifying my data was a key part of the project – and one were I restarted part way through, as can be seen in the commit messages[40] on GitHub around March 26th. For the first 600 tweets I hand-labelled. I simply decided on positive/negative. However, this proved to be very difficult, as often, no obvious sentiment was present, and other times, a tweet would have a very negative sentiment, but still be pro-vaccines.

This was problematic as the point of the study for me wasn't to classify simply the sentiments of Tweets, but to classify sentiment towards Vaccines specifically. A tweet that had a negative sentiment overall, but a positive outlook about vaccinations needed to be classified as positive – not negative.

As a result, I abandoned the Tweets I had classified, and started over (as seen in the commit messages). This time around, the classification methodology I adopted while labelling the training data was: "Does this tweet show that the person who wrote it is pro-vaccine, anti-vaccine, or can I not tell?"

Upon adopting this, I also changed my classification categories from just positive and negative, to positive, negative, and neutral. The process of preparing the training data took about 5 days – as I would label 1000 tweets a day, or 100 tweets from 10 states out of 50 total.

Moving on from preparing the training data. A slight modification of the same script was also used to split the tweet data not being used for training into individual text files as well. This script omitted the first 100 tweets in each .txt, as these tweets had been used for training, and as such would not be re-used for classification by the model later, this very slightly modified script can be found in *Appendix N*.

5.2.3 Data Pre-Processing

The data pre-processing stage was an essential part of the project that involved preparing the individual text files and text data that had been cleaned and formatted. Pre-processing involved several steps, summarized briefly as:

1) Remove all special characters
2) Remove all single characters
3) Remove single characters from start
4) Replace multiple spaces with single spaces
5) Removing pre-fixed "b" from loading function
6) Converting to lowercase
7) Lemmatization
8) Converting Text to Numbers
    a. Done using "Bad of Words" Model.
9) Removing stopwords
10) Finding TFIDF – (Term Frequency – Inverse Document Frequency)

As described earlier in the *Design* section, the pre-processing stage required the use of the Python "re" (regular expressions/RegEx) library, as well as the external Natural Language Toolkit (NLTK) library.

The steps involved in pre-processing occur in a number of files across the program, including when training the model (*see Appendix O*), evaluating the model (*see Appendix P*), and using the model to classify new data (*see Appendix Q*). This was because in all these cases, the new data that was

loaded in needed to be pre-processed before being handed to the Machine-Learning model.

The implementation for the "Pre-Processing" steps 1, 2, 3, 4, 5 and 6 was all done primarily using regular expressions and can be seen in *Figure 9*.

```
from nltk.stem import WordNetLemmatizer

stemmer = WordNetLemmatizer()

for sen in range(0, len(X)):
    # Remove all the special characters
    document = re.sub(r'\W', ' ', str(X[sen]))

    # remove all single characters
    document = re.sub(r'\s+[a-zA-Z]\s+', ' ', document)

    # Remove single characters from the start
    document = re.sub(r'\^[a-zA-Z]\s+', ' ', document)

    # Substituting multiple spaces with single space
    document = re.sub(r'\s+', ' ', document, flags=re.I)

    # Removing prefixed 'b'
    document = re.sub(r'^b\s+', '', document)

    # Converting to Lowercase
    document = document.lower()

    # Lemmatization
    document = document.split()

    document = [stemmer.lemmatize(word) for word in document]
    document = ' '.join(document)

    documents.append(document)
```

Figure 9: Data Pre-Processing Steps

Step 7 was "lemmatizing" the data. In lemmatization, words are reduced into the words dictionary root form. For example, "elephants" would be converted into "elephant". This is done in order to avoid creating features that are semantically similar but syntactically different. For instance, we don't want two different features named "elephant" and "elephants", which are semantically similar, therefore we perform lemmatization[10].

For the purposes of implementation, lemmatization was achieved using the "WorldNetLemmatizer" provided by the Natural Language Toolkit library, the import as well as implementation for this can be seen in *Figure 9*.

Step 8 was to convert the text data into numbers. Since the machine learning model cannot understand raw text, only numbers. There are a number of different approaches to doing this, including the World Embedding Model[41] and the Bag of Words Model[42]. For the purposes of this project, I used the Bag of Words Model.

The implementation for the bag of words model was done using the CountVectorizer function from Scikit-Learn. This can be seen in *figure 10*.

```
50  from sklearn.feature_extraction.text import CountVectorizer
51  vectorizer = CountVectorizer(max_features=1500, min_df=5,
    max_df=0.7, stop_words=stopwords.words('english'))
52  X = vectorizer.fit_transform(documents).toarray()
```

Figure 10: Bag of Words Implementation

There are some important parameters to be considered when using the CountVectorizer function. The first parameter is "max_features". When you convert text to numbers using bag of words. All the unique words in all the input text are converted into "features". Thousands of input tweets can contain tens of thousands of unique words. However, words which don't occur very often are usually not a good parameter for classifying. So, we use max_features to use only the 1500 most occurring words as features for our classifier.

Other parameters include min_df, and max_df. This corresponds to the minimum and maximum number of tweets a word can occur in. So, we only include words that occur in at least 5 tweets, and words that occur in less than 70% of tweets. Words that occur in almost every single tweet are not useful for classification as they tend not to provide any unique information about the tweet.

Lastly, the CountVectorizer is also passed a stop_words parameters defined as stopwords.words('english'). Stop words are words which are filtered out before the processing of natural language. There is, however, no universal list of stop words, nor any agreed upon rules for identifying stopwords[43]. For the purposes of this project, I used the "English" stopwords list provided by the Natural Language Toolkit, this completes step 9, leaving just step 10.

The one drawback of the Bag Of Words approach is that it assigns scores to words based on occurrence in one particular tweet. It doesn't take into account that the word in question may also occur frequently in other tweets as well. Step 10, or "TFIDF" solved this issue by multiplying the term frequency of a word by the inverse document frequency. While this sounds complicated, it is handled by the TfidTransformer function from Scikit-Learn. The implementation of which can be seen in *figure 11*.

```
54  from sklearn.feature_extraction.text import TfidfTransformer
55  tfidfconverter = TfidfTransformer()
56  X = tfidfconverter.fit_transform(X).toarray()
```

Figure 11: TFIDF Implementation

5.3 Data Classifying

After all the previous steps, the actual act of training the model, and then using the model to classify data was actually very simple.

```
classifier = RandomForestClassifier(n_estimators=1000, random_state=0)
classifier.fit(X_train, y_train)

with open('TweetClassifierModel.pickle', 'wb') as picklefile:
    pickle.dump(classifier, picklefile)
```

Figure 12: Training and saving the Machine-Learning Model

As seen in *figure 12*, the "classifier" was defined as using the Random Forest Algorithm, as discussed in the *Design* section, and set to have n_estimators of 1000, this just means the algorithm was set to have 1000 "Trees" in the forest. While more "Trees" would increase accuracy, it does so with diminishing returns and higher processing times, so 1000 was the number I settled on.

Following this, a simple .fit() function was used to train the Model. In this case, X_train holds the actual data from the tweets (however, this is not held as text, since it has been pre-processed, as discussed previously) and y_train holds the supervised signal label names (positive, negative and neutral). So that the Model knows what to associate each tweet with during training.

Once the model has been trained, we use the base Python "Pickle" library to save the model to a file, allowing us to load it and use it for classifying without having to re-train it each it.

Loading the model and using it for training can be seen in *Figure 13*.

```
with open('TweetClassifierModelEqualized.pickle', 'rb') as training_model:
    model = pickle.load(training_model)

y_pred = model.predict(X)
```

Figure 13: Loading the model, and using it to predict.

The model saves its prediction in the NumPy array y_pred. During implementation, this caused some headaches at the y_pred array was seemingly randomized, making it hard to track that classification back to an actual tweet (as the array itself simply held a 0, 1, or 2 for negative, neutral and positive respectfully). However, this turned out to not be an issue with the prediction, but with the load_files() function discussed earlier than was used to load the data (held in variable X).

The load_files function by default shuffles the data as it is loaded, which was something I did not realize and took far to long to figure out. This was eventually solved simply by adding a "shuffle=false" parameter (as can be seen in the code, available at *Appendix Q).*

The "for" loop that can be seen at the bottom of the Classifier.py script (*Appendix Q*) was used to create a three-level folder substructure for storing the classified data as positive, negative, or neutral for each individual state. This was done to have the data be in an easily readable fashion for the final part of the implementation. The front-end React.js App used to display the results of classification.

5.4 Data Presenting

The final part of the project implementation was a front-end consisting of a React.js Web Application that would be used to display the data in a graphical, easily accessible format. While this isn't explicitly related to the technologies and skills used elsewhere in the project. I had decided early on that this was something I wanted to do, as this would set aside my study on Vaccine Hesitancy from previous academically focussed studies.

As discussed in the *Design* section, the website makes use of Bootstap[32], Chart.js[12], React-Router[33], React-Twitter-Embed[34], React GitHub Pages[35] and React-Select[36].

First and foremost, while the website was developed for, and is best displayed on a large screen, responsive web development is a standard in these days. I took this account during development by using exclusively Bootstrap components for styling.

This provided multiple benefits – all Bootstrap components are responsive out of the box and all Bootstrap components come pre-styled. This second factor was also massively beneficial as I myself am no savant at UI or styling, so allowing Bootstrap to handle it was the easiest implementation.

The use of Bootstrap components can be found in any of the JavaScript (not TypeScript) files of the websites source code, contained in *Appendix R.*

All of the data on the website is visualized using Chart.js, this means that the Charts and Graphs are all dynamically generated by the page at runtime. This was an important factor during implementation as future extendibility was a key concern – I did not want the website to be set up using screenshots or images of graphs. It had to be set in such a way that it could be easily updated if more data was processed.

```
import StateData from "../assets/jsons/StateSentiment.json";
```

Figure 14: State Data Dynamic Load

As seen in *figure 14* this was done by loading and parsing data from a .json file and feeding that data to the Graph.js code. An example implementation of this is pictured in *figure 15* and *figure 16* where data is loaded and used for the Employment Rate comparison.

```
StateData.sort((a, b) => parseInt(a.UnemploymentRank) - parseInt(b.UnemploymentRank))
const labels = StateData.map((data) => (data.ID))
const pSentiment = StateData.map((data) => ((parseFloat(data.Positive)/400)*100).toFixed(2))
const nSentiment = StateData.map((data) => ((parseFloat(data.Negative)/400)*100).toFixed(2))
const tPSentiment = StateData.map((data) => (data.tPositive))
const tNSentiment = StateData.map((data) => (data.tNegative))
```

Figure 15: Loading Data for Comparison - and converting to %

```
const MLData = {
  labels,
  datasets: [
    {
      label: '% Positive Sentiment, ordered by highest Unemployment Rate
      on the left, lowest on the right. (As classified by the ML Model)',
      data: pSentiment,
      borderColor: 'green',
      backgroundColor: 'green',
    },
    {
      label: '% Negative Sentiment, ordered by highest Unemployment Rate
      on the left, lowest on the right. (As classified by the ML Model)',
      data: nSentiment,
      borderColor: 'red',
      backgroundColor: 'red',
    },
  ],
}

const ManualData = {
  labels,
  datasets: [
    {
      label: '% Positive Sentiment, ordered by highest Unemployment Rate
      on the left, lowest on the right. (As classified by Hand)',
      data: tPSentiment,
      borderColor: 'green',
      backgroundColor: 'green',
    },
    {
      label: '% Negative Sentiment, ordered by highest Unemployment Rate
      on the left, lowest on the right. (As classified by Hand)',
      data: tNSentiment,
      borderColor: 'red',
      backgroundColor: 'red',
    },
  ],
}

if (props.ML == "Yes") {
  var data = MLData
} else {
  var data = ManualData
}

return <Line options={options} data={data} />;
}
```

Figure 16: The data being passed to the graph being dynamic - using variables loaded from the .json

Part of making sure the website was easily extendable in the future was cutting down on extra code. A key part of React.js (and all JavaScript frameworks) is reusability. Parts of the website that look the same should be generated dynamically at runtime from the same source file (these source files are known as “components”).

This was a feature that I kept in mind during implementation to prevent the number of source files spiralling out of control. A key example of this is that users are able to view individual data from all 50 different states. This equates to 50 different pages when actually browsing. However, these 50 pages are generated using only 2 source files – one for the page itself and one TypeScript file for all 100 potential pie charts.

A few other novel implementations were also used for the website, the drop-down menu used to select which states individual data to view was implemented using the React-Select library and was extended and tested to only display the button to view an individual states data when a state was fully selected – either via the dropdown or by typing out the name of the state. This was done in a fairly “hacky” way, by using the conditional rendering functionality provided by React.js and combining it with an onChange hook available from the React-Select library. This solution, although probably not the best way to implement such functionality, is pictured in *figure 17* and can be found in the StateSelect.js file in *Appendix R*.

```
function StateSelector() {
    const [selectedValue, setSelectedValue] = useState(50)
    const handleChange = e => {setSelectedValue(e.value)}
    if (options[selectedValue].label == null) {
        var buttonLabel = "Please Select A State"
    } else {
        var buttonLabel = `View Data For: ${options[selectedValue].label}`
        var selectedState = parseInt(selectedValue) + parseInt(100)
        selectedState = '/' + selectedState
    }
    return(
        <Container>
            <Card>
                <Card.Body>···
                </Card.Body>
                <CardGroup>
                    <Card>···
                    </Card>
                    <Card>
                        <Card.Body>
                            <Select
                                value={options.find(obj => obj.value === selectedValue)}
                                maxMenuHeight={180}
                                options={options}
                                onChange={handleChange}
                            />
                            <div style={{
                                height: '80%',
                                display: 'flex',
                                alignItems: 'center',
                                justifyContent: 'center',
                                gridTemplateRows: 'auto 1fr'
                                }}>
                                {
                                options[selectedValue].label
                                    ?
                                        (
                                            <Link to={selectedState}>
                                                <Button variant="primary">{buttonLabel}</Button>
                                            </Link>
                                        )
                                    :
                                        (<div />)
                                }
                            </div>
```

Figure 17: State-Selector dropdown Implementation

Another novel part of the website worth mentioning is the “Tweet Display” feature at the bottom of each of the individual state pages. This feature shows the user a negative, neutral and positive tweet from the specific state they are currently browsing. The tweets being displayed are from the training sets used to train the machine learning model – as an example of what type of tweet would have been classified into each category.

This “Tweet Display” feature is also dynamically rendered each time an individual state page is loaded – and a random tweet from each states negative/neutral/positive pool is displayed. The Tweets themselves are pulled live from the Twitter servers using the unique ID of the tweet in conjunction with React-Twitter-Embed package. Not from reading the raw text in the .txt files generated during the *Data Formatting* stages. This was done to allow the Tweet

to be displayed as it originally would be seen on Twitter – and to follow Twitters stringent style guide and policies for how Tweets must be displayed.

The idea of wanting to randomly display a tweet from the negative/neutral/positive pools for each state caused a significant problem during implementation. The "tweet IDs" that would be used to pull the tweet itself from the twitter servers were stored in the names of files – not in the content of the files. This can be seen in the files in *Appendix M*.

This became a problem as the application was a Web App, rendered in browser – not server side (for example on a Node.js environment). The downside to being a Web App is that browsers don't have access to the local file system (this isn't actually explicitly true – browser-based file system access has some implementations, but none could do what I required). Due to not being able to access the file system, I couldn't ask the Web App to "randomly pick" a tweet from a certain states negative/neutral/positive pool to be displayed.

In the end, this was solved using the recursive "tree[44]" command in Linux to generate a JSON representation (*see Appendix S)* of the directories and files I needed to access – and pulling the data from that JSON, rather than from the file system and file itself. The code implementation of this can be seen in *figure 18* and *figure 19*, as well as in the TweetDisplayer.js file in *Appendix R*.

```
function TweetDisplayer(props) {
    var negative = ExampleTweets.contents[props.stateID].contents[0].contents
    var neutral = ExampleTweets.contents[props.stateID].contents[1].contents
    var positive = ExampleTweets.contents[props.stateID].contents[2].contents

    var negativeIDs = []
    var neutralIDs = []
    var positiveIDs = []

    for (let i = 0; i < Object.keys(negative).length; i++) {
        negativeIDs.push(negative[i].name.slice(-23, -4))
    }

    for (let i = 0; i < Object.keys(neutral).length; i++) {
        neutralIDs.push(neutral[i].name.slice(-23, -4))
    }

    for (let i = 0; i < Object.keys(positive).length; i++) {
        positiveIDs.push(positive[i].name.slice(-23, -4))
    }

    var randomNeg = Math.floor(Math.random() * negativeIDs.length)
    var randomNeut = Math.floor(Math.random() * neutralIDs.length)
    var randomPos = Math.floor(Math.random() * positiveIDs.length)
```

Figure 18: Pulling a Random Tweet ID for display, from the relevant states negative/neutral/positive pools

```
<TwitterTweetEmbed tweetId={negativeIDs[randomNeg]} />
```

Figure 19: Using React-Tweet-Embed and the variables defined in *figure 18* to display the tweets.

The final part of the *Data Presenting* was wanting the website to be fully deployed. This was done using React GitHub Pages. A key piece of information to note here is that GitHub Pages is designed for static web pages – and the "Create React App" command I used to generate my project is designed for single page web application.

This leads to two key problems – the website is neither static, nor is it single page. The dynamic-to-static conversion is handled by React.js itself during compile time, and the deployment by the React GitHub Pages[35] package.

However, the "multiple-page" nature of the website is essentially just a visual trick. Using React-Router[33] and conditional rendering – the website looks, feels, and browses as if it has multiple pages – in fact, even the URL changes. However, in actuality the web application is still a single page, just making use of extensive dynamic rendering to show and hide components based on what "page" the user wants to browse. The implementation of this can be seen in the "App.js" file, present in *Appendix R*.

6 Evaluation / Testing

The project was evaluated and tested in two ways, due to the two distinct parts of the project. The back-end was tested and evaluated internally during development, user-testing for this part wasn't possible as users would never interact with the back-end. As a result – user testing was implemented in the form of gauging interest in the project, seeing if this was actually something people cared about.

The second part of the project was the website, or the front-end, while this was also tested internally during development to make sure it ran robustly, I did release builds of the website to external testers to collect feedback and feature requests.

6.1 Back-End Evaluation

The back-end evaluation came down to how well the Machine-Learning model was able to classify tweet data, that was after all the heart of the project.

6.1.1 Unbalanced Training Data

Initially, the model was trained using a data-set of 5000 tweets (*see Appendix M)*, as described in the *Implementation* section of the report.

How the model's accuracy improved compared to the amount of data it was trained with can be seen in *figure 20*, including an extended extrapolation using a logarithmic curve. The solid blue line is the real evaluated accuracy, with the dotted blue line showing an extrapolation of the accuracy we can expect.

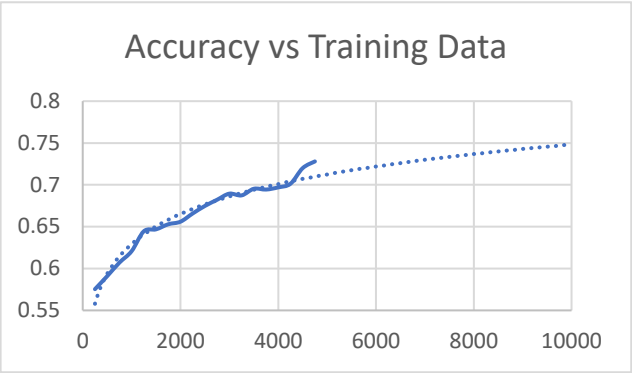

Figure 20: Accuracy vs. Training Data

The extrapolation curve was provided by Excel, and was calculated to have the following formula:

$$Accuracy = 0.0516ln(x) + 0.2729$$

Where "x" is the amount of training data provided to the machine learning model.

If I used all the training data I had available, the model could be expected to be about 72% accurate. However, this turned out to be an ineffectual model, regardless of the reported 72% accuracy number. The reason for this being that the training sets used to train the model were unbalanced. The training set was not split 33% each in terms of negative, neutral and positive data.

The effects of this can be best explained using a confusion matrix, for this model, the confusion matrix can be seen in *figure 21*.

	Actual Negative	Actual Neutral	Actual Positive
Predicted Negative	27	56	74
Predicted Neutral	6	372	98
Predicted Positive	0	106	293

Figure 21: Confusion Matrix for Unbalanced Training Set

To explain the data above, we'll consider two categories, from which we can derive two factors that we can use to judge the model. The two categories being the number of correct predictions vs. actual occurrences (this constitutes precision), and the number of mislabelled predictions (this constitutes accuracy)

Firstly, to look at the precision:

- The model correctly predicted 27 negative occurrences, out of a potential 33
- The model correctly predicted 372 neutral occurrences, out of a potential 534
- The model correctly predicted 293 positive occurrences, out of a potential 433

This means the model had a precision rating of 81%, 70%, and 68% across negative, neutral, and positive respectfully.

Now, to look at accuracy:

- The model mislabelled 6 negative as neutral, and 0 negatives as positive
- The model mislabelled 56 neutrals as negative, and 106 neutrals as positive
- The model mislabelled 42 positives as negative, and 98 positives as neutral

This means the models accuracy rating was 22%, 78% and 73% across negative, neutral and positive respectfully.

To figure out if the model is indeed effective, we need to compare these figures against two baselines, a majority baseline, and a random baseline.

The majority baseline is a simple one to test against – the majority of training data that the model received was neutral. So, a majority baseline would expect the model to predict everything as neutral. This wasn't the case, and as such the model does beat a majority baseline.

The random baseline entails that if the model was to simply predict randomly the sentiment of a tweet, how likely is it to be correct? The set the model was tested against was equalized, so there would be a 33% chance the model would be correct if it was simply predicting randomly.

The model beats this 33% across the board for precision, however, it fails to beat it when it comes to accuracy. As the 22% accuracy number for negative predictions is far worse than what we'd expect if the model was simply predicting randomly. The reason for this failure is also clear, the model received far more training data for neutral and positive tweets than it received for negative tweets, as the training sets were unbalanced. As such, it's predictions were skewed away from negative, and towards positive and neutral. This resulted in a model that could not be used effectively.

6.1.2 Balanced Training Set

To solve the issues with the model, the training data set needed to be balanced, and so it was, this balanced training set can also be found in *Appendix T*.

The big downside to needing to equalize the training data was that I would be losing more than half of the 5000 tweets I had initially labelled. This was due to the fact that in the initial labelling, only 691 tweets were labelled as negative. As a result, both the positive and neutral categories would be restricted to 691 tweets as well. This led to a total of 2073 pieces of training data, a far cry from the original 5000.

As a result of this, as *figure 20* shows us, we could only expect the model to be around 67% accurate. While this is disappointing, it is entirely down to the drop in training data, and as a result, can easily be rectified in the future or given more time – as the only thing necessary to improve is simply classify more data for training.

Regardless, we still needed to check if the new model was effectual, this can be done in the same way as with the unbalanced model.

The confusion matrix for the model training with a balanced training set can be seen in *figure 22*.

	Actual Negative	Actual Neutral	Actual Positive
Predicted Negative	95	20	27
Predicted Neutral	18	80	26
Predicted Positive	17	37	95

Figure 22: Confusion Matrix for Balanced Training Set

Similar to before, we need to now consider both precision and accuracy.

Firstly, to look at the precision:

- The model correctly predicted 95 negative occurrences, out of a potential 130
- The model correctly predicted 80 neutral occurrences, out of a potential 137
- The model correctly predicted 95 positive occurrences, out of a potential 148

This means the model had a precision rating of 73%, 58%, and 64% across negative, neutral, and positive respectfully.

Now, to look at accuracy:

- The model mislabelled 18 negative as neutral, and 17 negatives as positive
- The model mislabelled 20 neutrals as negative, and 37 neutrals as positive
- The model mislabelled 27 positives as negative, and 26 positives as neutral

This means the models accuracy rating was 67%, 65% and 64% across negative, neutral and positive respectfully.

If we were to now compare this model where the previous one failed – against the random baseline, we see that it beats the expected 33% accuracy and precision of a random baseline across all categories.

As a result, although the overall accuracy of 67% was lower than with the unbalanced training set, this was the model that was used to classify the data that is seen on the website.

6.1.3 Back-End User Testing

As mentioned earlier, user testing for the back end wasn't possible in the traditional sense. So instead, I conducted interviews with people to see if they were interested in what I was trying to do, after all, what's the point in the study if no one cares about the results.

I did this in the form of interviews, while making sure to follow the required ethical procedures, and obtaining the necessary informed consent forms from users.

During the interviews, I took notes in my personal notebook, making sure that none of the data recorded was personally identifiable.

After all this procedure, I was quite glad that in all the interviews I conducted, the interviewees unanimously agreed that this project was something they thought was worthwhile, and something they would be interested in seeing the results off.

6.2 Front-End Evaluation

The second part of my project was much easier to evaluate and test in a traditional sense when it came to external user testing.

After reaching what I considered to be a "finished" state with the website, I invited people to test it in an "interview" setting, as I feel it's most valuable to be able to see how a user uses the website, and what they are drawn too. This also allowed me to ask any questions that may come to mind or be relevant in the moment, rather than restricting the feedback to a survey.

During user testing for the website, a few things were noted and changed. Firstly – users complained about the bright color scheme of website. One user even used the word "flash-banged", this led to a number of changes with the background color, moving away from a traditional white background and instead opting for gradients of grey, to

emulate a sort of "dark mode" and make the browsing experience more pleasant.

Beyond this, users did not have any complaints about the website, or any requested changes, however, one user did request an additional feature. At the time the website was tested, it was not possible to browse data for individual states.

I agreed with this feedback, and it led to the development and implementation of the "individual state pages", as well as the ability to view example tweets from each states negative, neutral, and positive pools.

On development of these pages, I invited that same user back to get their opinion. The feedback received on this was positive, with one suggestion being made – now that I was displaying raw tweets, the user recommended I add a disclaimer for potential foul language that may be present. This disclaimer can now be seen before users view the individual state pages, to server as a warning.

One user that tested on a mobile device complained about a responsiveness issue present with the navbar/header – however, this was an issue that I deprioritized as time was tight, and as result it was never addressed. It's important to note that the website still works well on mobile and is fully responsive – this was just a single issue with the navbar/header.

7 Description of the final product

The final product is twofold – both a back-end and a front-end.

The back end is not user facing, and has no UI or such, however, all the code, all the script and functions that have been written or used have been designed to be scalable.

The cURL script used to pull data from the v1.1 API is available and can be re-used. The scripts used to clean, format, structure, and pre-process the tweet data provided from the API are both available and once again designed to be easily used with any extra data that a future extension of the project may want to add.

Furthermore, the "pickled" version of both Machine-Learning models trained with the balanced and unbalanced datasets can be found in *Appendix U*.

Essentially, as far as a "final product" for the back end is concerned, all the code and methodologies are available, as well as being designed to be easily extendable for future use.

The front-end website is a more traditional, user facing, final product. The website is a simple, easy to navigate web application that allows users to explore the classified data from the study.

The website further puts this data up against 6 different statistics in an attempt to find any potential correlation. The 6 statistics being:

- State Political Leaning
- State Public Health Spending
- State Literacy Rate
- State Employment Rate
- State Average Age
- State Average Salary.

While no correlations can be seen with the current classified data, this is highly likely due to the performance level of the model – which as discussed can be easily improved simply with more training data.

Due to the performance level of the model – the website places all the data classified by the model next to the data that was classified by hand, allowing users to see how the model is different, and to compare and contrast.

The website lastly also allows users to browse data for individual states, as well as viewing the types of tweets that had been classified as positive, negative, and neutral during the training process.

The website is live, and can be browsed at https://griffinkf.github.io/Y4-HonorsProject/ (due to the nature of GitHub Pages, and the "hacky" way the website has been deployed as a dynamic site on a service meant for static pages, the URL is actually case sensitive)

8 Appraisal

If I was to appraise my own feelings towards the project in one word, I would settle on "frustrated".

The reason I choose frustration as my primary feeling towards the project is because I think the steps taken, the methodologies followed, the code written and all the work done was done correctly, however, at the end of the day. The model only managed to be around 67% accurate, and this meant it was not usable to be able to draw correlations when compared to other statistics – which was what I was hoping to achieve. After all, the project was titled "Data Mining Human Reasoning" and the goal was to gain an understanding of why people have a certain vaccine sentiment. Ultimately, this goal was not achieved.

With a goal not being achieved, one might ask why I say frustrated rather than disappointed. This is primarily because although the goal was not achieved, I don't feel that I failed because I was incorrect in my approach. In all honesty, simply having a more accurate model would allow the goal to be achieved, and a more accurate model would

very simply be achieved by more training data (as *figure 20* suggests).

More training data is simple to obtain, however it is time consuming. As an individual attempting this project, only so much time can be allocated to reading tweets and labelling them for training purposes – as such a task is neither technical nor does it display any level of competence. Regardless, the reality is that this boring, simple, and grindy task makes or breaks the final performance of the model, and by extension the ability to draw meaning or conclusions from the data.

If I was to do this entire project again, the first and foremost thing I would do differently is in my approach to Twitter, and the Twitter API. The truth to accept is that the access level permissions I had for the Twitter API simply were not suited to collect the data I needed with the details I needed.

Although I primarily think that useful conclusions are unable to be drawn from the data due to the models performance, it's also a point to note that if the model was 100% accurate – the sentiment of a state is still not actually entirely from people in that state. As I was unable to access geographical data, only mentions of state names.

Someone from Arkansas could easily be tweeting about New York and vice versa, and for a topic as divisive as vaccinations, people from every corner love to chip in about states that they may never have set foot in.

So, the primary thing I'd do differently is only attempt something like this if I was able to obtain proper research level permissions to the v2.0 endpoints, as without that level of access, I was hamstring right from the start.

However, in terms of technologies used, and the approach otherwise taken, I would not change anything. Reflecting on the project plan, at a high enough level, implementation went as expected. Only at a more granular level did plans have to be changed, but, as discussed in *Specification* and in *Design*, this was something I was keenly aware of and anticipated on the way in.

As a final note, the most useful learning aspect for me in this project, and one that I will carry forward to any future projects of this scale is understanding and limiting the things that are within, and outside of my own control.

A bug in a program I am writing is something that is in my control, although it may be frustrating or hard to solve, at the end of the day it is possible.

Being unable to access certain data points from an API, or getting suspended due to being caught in a spam filter, or any number of the issues I had with Twitter are all primarily things that are outside of my own control. So, for future projects, I think it's important to take into account and allocate a large amount of extra time for trying to deal with things that are unexpected, and outside of your control. Or, even aside from extra time, to consider the overall viability of a project if large parts of it are not something I myself can control.

9 Summary and Conclusions

To summarize the main point of my project is simple – An Attempt to Understand Vaccine Hesitancy in the United States of America.

While this point may not have been achieved, the work put towards it is certainly in the right direction, and this report provides an adequate guide for anyone who may want to take on such a project themselves. With the pitfalls and areas to watch cleanly laid out.

As discussed in *Appraisal*, although no conclusions or outcomes were taken away from this project explicitly regarding understanding vaccine hesitancy, that doesn't mean nothing was learned.

10 Future Work

One of the key areas in which this project was a success is how it leaves the door open for future work. As discussed in *Description of Final Project*, the work done here is both available, and designed to be easily extendable.

Even just the addition of more training data would immediately yield cascading positive results across the rest of the project.

Although a seemingly simple addition to improve the project in the future would be to supplement the supervised machine learning with lexicon analysis as well, this is something that I researched and not a path I would recommend going down.

As discussed, the tweets were not classified explicitly considering the raw sentiment of the tweet. The classification was instead done considering the sentiment towards vaccines the author of the tweet has – based on the information available in the tweet. This level of complexity beyond the simple text of the tweet would make it impossible to effectively define a corpora or dictionary or words to then use for lexicon analysis.

It should also be noted that I initially had the idea that would seed this project back in my second year of university – more than two years before ever doing this project. The idea has lived in my head since then, and the area of vaccine hesitancy is something that I continue to have a keen interest in. Combine that with primarily feeling frustrated at the shortcomings of the project – it's not out of the realms

of possibility that I continue to extend the work I have done here in my own time.

Acknowledgments

While I think it is natural and almost automatic to acknowledge the project supervisor. I'd like to do so while emphasizing that I'm acknowledging the help provided by Prof. John Lawrence in least token way possible. There more than one period of time during the project when I felt I was at a dead end, only for help to be given and progress to continue, for this I can only say: Thank you, Sir.

On a less serious note, I'd like to thank the Ryzen 5950X and all 16 of its cores that are present in my desktop PC. As the level of data to be classified increased, so did the temperatures, but it held steady and never made me wait too long.

I'd also like to acknowledge Twitter, the Twitter API, and the Twitter Developer program, who claim to love developers. To which I can only say: I'm not so sure about that one.

References

[1] Durbach, N. They might as well brand us: Working class resistance to compulsory vaccination in Victorian England. The Society for the Social History of Medicine. 2000;13:45-62.

[2] Article on Unvaccinated Patients: https://northeastlondonccg.nhs.uk/news/almost-90-of-patients-admitted-to-intensive-care-units-innorth-east-london-are-not-fully-vaccinated/

[3] Cambridge Analytica Swinging Elections: https://www.theguardian.com/news/2018/may/06/cambridge-analytica-how-turn-clicks-into-voteschristopher-wylie

[4] Piedrahita-Valdés H, Piedrahita-Castillo D, Bermejo-Higuera J, et al. Vaccine Hesitancy on Social Media: Sentiment Analysis from June 2011 to April 2019. Vaccines (Basel). 2021;9(1):28. Published 2021 Jan 7. doi:10.3390/vaccines9010028

[5] Almatarneh, Sattam, and Pablo Gamallo. "A lexicon based method to search for extreme opinions." PloS one 13.5 (2018): e0197816.

[6] VADER: https://github.com/cjhutto/vaderSentiment

[7] Neha Puri , Eric A. Coomes , Hourmazd Haghbayan & Keith Gunaratne (2020) Social media and vaccine hesitancy: new updates for the era of COVID-19 and globalized infectious diseases, Human Vaccines & Immunotherapeutics, 16:11, 2586-2593, DOI: 10.1080/21645515.2020.1780846

[8] Muric G, Wu Y, Ferrara E. COVID-19 Vaccine Hesitancy on Social Media: Building a Public Twitter Data Set of Antivaccine Content, Vaccine Misinformation, and Conspiracies. JMIR Public Health Surveill. 2021 Nov 17;7(11):e30642. doi: 10.2196/30642. PMID: 34653016; PMCID: PMC8694238

[9] Twitter API: Twitter API Documentation | Docs | Twitter Developer Platform

[10] Stack Abuse: Text Classification with Python and Scikit-Learn: https://stackabuse.com/text-classification-with-python-and-scikit-learn/

[11] React.js, A JavaScript Library for building user interfaces: https://reactjs.org/

[12] Chart.js, Simple yet flexible JavaScript charting for designers and developers: https://www.chartjs.org/

[13] SourceForge, Compare, Download & Develop Open Source & Business Software: https://sourceforge.net/

[14] Bitbucket, The Git solution for professional teams: https://bitbucket.org

[15] Git Lab, iterate faster, innovate together: https://gitlab.com/

[16] GitHub, Where the world builds software: https://github.com/

[17] Postman API Platform: https://www.postman.com/

[18] Postman API Twitter 2.0 Endpoints Collection: https://github.com/twitterdev/postman-twitter-api

[19] Microsoft Windows Subsystem for Linux (WSL): https://docs.microsoft.com/en-us/windows/wsl/about

[20] cURL, a command line library for transferring data: https://curl.se/

[21] Scikit-Learn: Machine Learning in Python: https://scikit-learn.org/

[22] Python Programming Language: https://www.python.org/

[23] NumPy, https://numpy.org/

[24] re, https://docs.python.org/3/library/re.html

[25] Natural Language Toolkit (NLTK), https://www.nltk.org/

[26] pickle, https://docs.python.org/3/library/pickle.html

[27] TensorFlow, an end-to-end open-source platform for machine learning: https://www.tensorflow.org/

[28] Angular.js: https://angular.io/

[29] Vue.js: https://vuejs.org/

[30] Frameworks Comparison Video, Fireship: https://www.youtube.com/watch?v=cuHDQhDhvPE

[31] Most Popular Frameworks, according to Statista: https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/

[32] Bootstrap, https://getbootstrap.com/

[33] React-Router: https://reactrouter.com/

[34] React-Twitter-Embed: https://www.npmjs.com/package/react-twitter-embed

[35] React-GitHub-Pages: https://github.com/gitname/react-gh-pages

[36] React-Select: https://react-select.com

[37] Covid Tweets Dataset from Kaggle: https://www.kaggle.com/datasets/kaushiksuresh147/covidvaccine-tweets

[38] Kaggle: https://www.kaggle.com/

[39] sklearn.datasets.load_files(): https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_files.html

[40] Link to Github Commit History set to March 26th: https://github.com/GriffinKF/Y4-HonorsProject/commits/main?before=ca99b369b74722275ff1898c965f2819f2c29282+70&branch=main

[41] Word Embedding Model: https://en.wikipedia.org/wiki/Word_embedding

[42] Bag of Words Model: https://en.wikipedia.org/wiki/Bag-of-words_model

[43] Stopwords: https://en.wikipedia.org/wiki/Stop_word

[44] Linux “tree”: https://linux.die.net/man/1/tree

Appendices

A. Hand-Written meeting log for first meeting with John Lawrence.
B. Gantt Chart for Semester 2
C. Example of JSON Returned by API
D. Email Containing Resources for Scikit-Learn provided by supervisor (John Lawrence)
E. Twitter Acknowledges Application Appeal
F. Twitter Returns API access
G. 7-Day Endpoint Result for Each State
H. Tweet Dataset Obtained from Kaggle
I. Premium Search Invoice
J. Link to Full Project Repository
K. Tweet Cleaning Script and Result
L. Manual Classifier Script
M. Training Sets
N. Tweet Splitter
O. Model Training
P. Model Evaluating
Q. Classifying New Data
R. Website Components Source Code
S. JSON used for Tweet Displayer.
T. Balanced Training Data
U. Pickled Models
V. Full Source Code
W. Project Poster