Abstract

Analyzing web articles related to crisis events can help social scientists gauge public sentiment and form public policy around how to react to such disasters. However, data collection for such tasks is difficult. Manual dataset curation is time-consuming and costly, as a user needs to use some sort of search engine to iterate through multiple web pages, painstakingly analyzing each document thoroughly to determine the crisis events it may be related to. Automated processes, however, such as web crawlers, operate primarily via rule-based methods, which may not accurately classify individual documents as being related to the crisis event of interest. In our work, we seek to use machine learning techniques to determine whether individual documents are related to a specific crisis event using natural language processing techniques. To accomplish this, we treat the area of interest as a single class, and consider all other topics as not being of interest. We hypothesize that natural language processing techniques can be used to classify a particular webpage as being relevant to a certain crisis. A potential motivation for this approach is to guide efficient web crawling using techniques from semantic analysis.
9 Data Files
9.1 HTML ....................................................... 33
9.2 URLs ...................................................... 33
9.3 Text files ................................................... 34

10 Deliverables ........................................... 34
10.1 Backend .................................................. 34
10.2 Frontend ................................................ 35
10.3 Accessibility ........................................... 35

11 Lessons Learned ...................................... 35
11.1 Training a one-class SVM ........................ 35
11.2 Future Plans ........................................... 35

12 Challenges .................................................. 36

13 Timeline ................................................... 36

14 Acknowledgements ................................... 36
List of Figures

1. Training a model from URLs ................................................. 7
2. Training a model from HTML documents ............................ 8
3. Training a model from plaintext ......................................... 9
4. Getting predictions from URLs ........................................... 9
5. Getting predictions from HTML documents ......................... 10
6. Getting predictions from text ............................................ 11
7. Home Page .................................................................. 12
8. Training Page ................................................................. 13
9. Labeling Page ................................................................. 14
10. Inference Page .............................................................. 14
11. Save Webpage ............................................................... 16
12. Compress HTML files ..................................................... 16
13. Upload a zip file of html webpages ..................................... 17
14. Specify the type of model to train ....................................... 17
15. Perform Training ............................................................ 17
16. Training Loss Curve ......................................................... 18
17. Copy the URL from the search bar .................................... 18
18. Compress URLs ............................................................... 19
19. Upload a zip file of URLs .................................................. 19
20. Specify the type of model to train ....................................... 19
21. Perform Training ............................................................ 20
22. Compress .txt files .......................................................... 21
23. Upload a zip file of text files ............................................. 21
24. Specify the type of model to train ....................................... 21
25. Perform Training ............................................................ 22
26. Labeling Threshold Slider ................................................ 23
27. Cosine Similarity Plot ....................................................... 24
28. Documents List ............................................................... 24
29. Compress HTML files ..................................................... 27
30. Upload a zip file of html webpages ..................................... 27
31. Specify the type of model to generate outputs with .......... 27
32. Perform Inference ........................................................... 28
33. Documents List ............................................................... 28
34. Copy the URL from the search bar .................................... 28
35. Compress URLs ............................................................... 29
36. Upload a zip file of URLs .................................................. 29
37. Specify the type of model to generate outputs with .......... 29
38. Perform Inference ........................................................... 30
39. Compress .txt files .......................................................... 31
40. Upload a zip file of text files ............................................. 31
41. Specify the type of model to generate outputs with .......... 31
42. Perform Inference ........................................................... 32
43. HTML Example file ......................................................... 34
1 Introduction

Binary text classification techniques split the dataset into two classes. While a possible approach to our problem would be to treat the topic of interest as one class in a binary classification problem and consider all “other” topics be the second class, this approach is prone to errors as the “other” class is not well defined. One-class classification approaches seek to identify elements of a target class from amidst a testing dataset containing multiple separate classes. Because they do not suffer from a poorly-defined “other” class, one-class classification techniques are the most suitable method for identifying crisis event web pages. Typically, one-class classification models are trained on a training dataset that contains only elements of the target class, though some adversarial approaches include multiple classes in the training data. The goal of the model is to learn an approximate mapping to a higher-dimensional Euclidean space that minimizes the distance between elements of the target class while maximizing distance from all “other” elements.

To expedite the model creation process and allow the client to quickly and efficiently create new one-class classification models using a variety of techniques and strategies, we develop a pipeline with a variety of useful features for both the training and inference aspects of the classification process. These features will hopefully allow our client to quickly train and perform inference on one-class classification models of various types, using a variety of different training strategies as well as different input and output data requirements. These feature requirements are described in more detail in the Features section.

2 Team Organization

We have two team members, each with a crucial role in the team composition:

- Caleb McIrvin - team lead, point of contact with the client, backend lead
- Abdurrehman Nauman - frontend developer, UI design lead

3 Features

Our pipeline has a number of required features to allow the client to effectively manipulate the data and rapidly iterate over model prototypes. These features can be broadly split into the training and inference categories, depending on the specific task the client wishes to accomplish.

3.1 Training

If the client is looking to train a specific model, they need to be able to input various types of data, depending on the specific dataset that they have. As a result, our pipeline needs to be robust enough to handle changes in input data. Different types of training input data we are working on supporting with our pipeline are given below.
1. Training a model from URLs

Our client needs to be able to train a specific one-class classification model by passing in a specific training set of data to our pipeline. By passing in the data to the pipeline and specifying that they want a model trained, the client can efficiently and effectively receive a well-trained model that they can then use in a downstream classification task, potentially as a subroutine in a web crawler designed to grab crisis events. In order for our client to easily train the model, we need to support a variety of input formats. The first format we support is a text file of URLs, separated by line. This type of file, typically around 20-50 hand-curated URLs, is simple to create for the client, but slightly more difficult for the pipeline to train on, as the pipeline would need to request the content at each specific URL to train the model on. This could cause latency issues when running the pipeline for the client, especially if the client has a poor Internet connection, but, as it requires the least effort on the client’s end, it may be a good choice for them.

![Figure 1: Training a model from URLs](image)

2. Training a model from HTML

If the client has, for example, a dataset of HTML files, then this can reduce the latency of the model creation process significantly, as the model no longer has to request the HTML data from the Internet. However, the backend web scraping scripts may still take significant amounts of time to run, as the HTML must be parsed in order to generate text for the model to train on. As HTML is a structured language consisting of numerous tags, such as the paragraph and div tags, none of which are useful training data for the model, the relevant textual data must be extracted from the HTML in order for the model to effectively train. As HTML web scraping is notoriously difficult due to inconsistent standards between different web developers in addition to a variety of JavaScript libraries leading to potentially vastly different code landscapes, the HTML parsing scripts may take significant amounts of time as
well. Additionally, it is significantly more difficult for a client to manually curate a dataset of HTML files in comparison to a dataset of textual URLs, as the client would need to perform multiple extra steps before sending the data to the pipeline.

Figure 2: Training a model from HTML documents

3. **Training a model from text documents**

This dataset is the cleanest dataset of the three input datasets our pipeline is designed to support, as very little preprocessing needs to be done. As the data is already in a textual format, the backend merely needs to finish the preprocessing stage by vectorizing individual documents to prepare for input to the model training script. Our approach to document vectorizing is described in greater detail in the Tech Stack section. This method, however, requires the greatest work from the client, as they would have to manually curate multiple web pages and extract the relevant text, which is tedious and time-consuming. However, our hope is that, once the initial dataset is curated, the trained classification model would make the process of classifying future documents significantly simpler, providing a key step in the automation process.

3.2 **Inference**

During inference, when the client wants to use the models they have generated using our pipeline, the process for using these models should be rapid and simple. During inference, the client should always pass in a pre-trained one-class classification model they want to use to classify specific documents. Additionally, our pipeline needs to be able to support a variety of test data formats for ease of use and accessibility. Various scenarios the client may run into are given below.
1. **Predicting class of input URLs**

If the client has a text file of URLs, each leading to a specific webpage, and the client wants to know whether each of these URLs are related to a specific crisis event or not, then this should be a straightforward process for the client to perform. Our pipeline needs to be able to read through each line of the input text file, request the HTML of the specific URL stored on each line, and then generate classification model output based on the content of the webpage at the specific URL. In order to accomplish this, the pipeline requires functionality that can efficiently and correctly parse a text file containing multiple URLs.
2. **Predicting class of input HTML files**
   The client also needs to be able to parse individual HTML files, or a zipped collection of HTML files. As this is a common way to store webpage content, this needs to be a feature that our pipeline possesses. If the client chooses to input a collection of HTML files, then the pipeline goes through fewer steps in relation to the input text file of URLs. Similarly to the training step, where the client could also input a set of HTML documents, the HTML parsing script takes a significant amount of time, as web parsing is generally a challenging problem.

![](image)

**Figure 5: Getting predictions from HTML documents**

3. **Predicting class of input text**
   While the simplest method for the model is to simply collect a piece of text and use the text for inference, it is more difficult for the client, as the burden of preprocessing shifts from the backend to the client. Additionally, inputting text into a text form on the frontend of the pipeline may be inefficient, as copying and pasting large quantities of input text can be cumbersome. As a result, the pipeline needs to support an input file of text, as opposed to simply a text submission form on the frontend.

4 **Motivation**

Understanding the public reaction to crisis events is crucial to creating public policy regarding events that can legitimately be termed as crises. In order to understand what sort of view the populace has on a specific crisis event and what type of impact that crisis event has had, data scientists need to be able to collect large quantities of web articles related to a specific crisis event they’re interested in studying. However, manually collecting this
data is a painstaking process and requires significant investment. As a result, automatic approaches are preferable, but suffer from an effective way to determine if a specific webpage is actually related to the crisis event of interest. Our solution, using one-class text classification models to determine relevance, seeks to alleviate this problem by classifying web pages efficiently and accurately, removing much of the manual effort required in classifying the web pages by hand.

5 Requirements

The end product will have the ability to take some training data and feed it into a one-text classification model in order to be able to create a web crawler that will be able to find relevant web pages. The classifier will be able to sift through relevant web pages to extract certain information about the event and estimate the probability that a given webpage contains relevant data. The end result will be the workflow and model training scripts that could be used to create classifiers for other crisis classifications. A website will be produced that would allow a user to input information (a list of URLs) about a crisis and the website will use the trained model and be able to find more relevant pages. The product would return the URLs to other web pages that the model decided are relevant to the topic the model was trained on.

6 Implementation

Our tech stack consists of three different pieces - the frontend, which is responsible for interacting with the client and determining which type of action the client is looking to perform, the backend, which actually performs the requested actions and is responsible
for the model creation and inference processes behind the scenes, and the application programming interface (API), which allows the frontend to communicate with the backend.

6.1 Frontend

The frontend’s primary goal is to handle interactions with the client. It does this through a JavaScript web interface with which the client can interact to achieve their specific tasks. For our web interface frontend, we used the Vue framework, as our team had worked with it before and had experience coding web applications in Vue.

During the training steps, the frontend needs to be able to take in a variety of different types of input data. To do this, there are a variety of different HTML elements we could potentially use in our application, including dropdown menus for selecting a specific input data type and upload forms to load in data and models.

Our frontend is split into four separate pages, the home, training, labeling, and inference pages, as described below.

6.1.1 Home Page

The home page serves as the entry point for users looking to use the application. When a user first opens up the application, they are greeted by the home page, which displays links to the other three parts of the application, the training, labeling, and inference stages. An image of the home page is shown below in Figure 7.

![Home Page Image](image.png)

Figure 7: Home Page
6.1.2 Training Page

On the training page, the user can train a new one-class classification model. The primary actionable items in the training page are the buttons to upload data and commence training the model. An image of the page layout is given in [8].

![Training Page](image)

Figure 8: Training Page

6.1.3 Labeling Page

When the user navigates to the labeling page, they are given the option to upload both relevant and non-relevant data as two separate zipfiles. In this stage, the user performs the labeling task described in greater detail in the user manual. The labeling page is pictured in [9].

6.1.4 Inference Page

The inference page is where the user can perform inference on previously unknown data and should be the final page of the application the user accesses. The layout is shown below in [10].

6.2 Backend

For our backend, we decided to use Python, as Python has excellent support for a variety of machine learning tasks. Because Python allows us to easily train new machine learning models and perform inference on them, it was a simple choice to use Python as our backend.

The primary role of the backend is to handle the data processing and model training components of the pipeline. For example, if we wanted to train a neural network using a
Figure 9: Labeling Page

Figure 10: Inference Page
text file of URLs, the backend would need to extract individual URLs from the text file, get the HTML content of each individual URL, parse the HTML to get the actual text of each document, vectorize the text, and feed the vectorized text to the model for training. These different backend components serve as required functionality for the required features as described in Section 2 to function correctly.

6.3 API

In order for the frontend to communicate with the backend, they need to go through an application programming interface, or API. This API standardizes communication flow between the frontend and backend, allowing them to listen on and post data to the same parts of the application. Typical applications often use a representational state transfer (REST) API, which defines a set of operations, including POST and GET, that the frontend and backend can use to define how a particular piece of data is supposed to interact with the rest of the application.

For our API, we use Python’s Flask library to set up an API server accessible to both our frontend and our backend. This server takes REST requests from both the frontend and backend and updates the state of each piece of the application accordingly. For example, consider the scenario where the client is trying to perform inference on a set of documents, given that he already has a trained one-class classification model. He would send the model and data to the frontend, which would then POST both the model and the data to the appropriate API endpoint, depending on the type of model and type of data. The backend would then GET the model and documents and begin using the model to classify the documents. When this step terminates, the backend would then POST the results from the model back to the API server, which would direct these results to the appropriate place in the frontend. Once the frontend receives the classifications, it would visualize the results for the client to see.

7 User Manual

Project users should refer to this section if they have questions regarding the proper use of our model training / inference application. As our application was designed with three separate stages in mind, model training, threshold generation, and dataset inference, we describe these three stages below in depth.

7.1 Use Environment

This application is designed to be used both to train, threshold, and generate predictions from one-class classification models that can accurately determine whether or not a document is related to a specific crisis event. We provide motivation for this task as a subroutine in a web crawler application used for dataset generation. This application is not designed to train a model without a dataset, but requires some initial manual dataset curation to train a model.
7.2 Training a Model

The application currently accepts three data formats - HTML webpages, URLs, or text data. To create a dataset for training a model, users should follow one of these three formats.

7.2.1 HTML Web Pages

Step 1
If this is the preferred data format, users should go to their web pages of interest. In a web browser, right-click on the webpage and click “Save As” to save the website as an HTML document, as seen in Figure 11.

![Figure 11: Save Webpage](image)

Step 2
Once you have saved all of your webpages, then use a zip utility to create a zipfile of all of the saved HTML web pages. Alternatively, on Windows or MacOS, you may be able to use the built-in zip utility by right clicking on all of the .html files and selecting Compress (zip on Windows), as pictured in Figure 12.

![Figure 12: Compress HTML files](image)

This should create a .zip file that you can upload to the web application.

Step 3
On the web application, use the “Choose Files” button under “Upload Training Data” and select the zip file from your local file system.

**Step 4**
Additionally, from the dropdown underneath Training Data Type, select the “Zipfile of HTML” option to tell the application that you are passing in a zipfile of HTML web pages.

![Figure 13: Upload a zip file of html webpages](image)

**Step 5**
Additionally, the user will need to specify the type of model to be trained by the application. In order to specify the type of model to be returned by the application, select the appropriate model type from the “Model Type” dropdown. We are currently working on expanding the number of model types supported.

![Figure 14: Specify the type of model to train](image)

**Step 6**
To start training a model, select the Perform Training button to send the data through the application. The application will then return a model that can be downloaded using the Download button.

![Figure 15: Perform Training](image)

**Step 7**
After training has completed, the application will display the loss curve of the model during the training process. A good loss curve indicating that the model learned the task well should show decreasing loss (y-axis) as training time increases (x-axis increases). A sample loss curve is shown in Figure 16.

Figure 16: Training Loss Curve

7.2.2 URLs

Step 1

If submitting a text file of URLs to relevant web pages is the preferred methodology for uploading model training data, then the user should first compile a list of URLs to train the model on. This can be accomplished by creating a text document using a text editor such as TextEdit for Mac and opening a new .txt file. Note that Word (.docx) and Pages (.pages) files will not work for the application at the moment, we are working on adding support for these formats in the future.

Once the text file has been created, the user should add URLs to web pages related to the crisis event of interest to the document. To add a particular URL to the document, go to the web page, select the URL from the search bar, copy the URL from the search bar, and paste the URL in the text document, as seen in Figure 17.

Figure 17: Copy the URL from the search bar

Step 2

Once the .txt file of URLs has been created, then use a zip utility to create a zipfile of the .txt file. Alternatively, on Windows or MacOS, you may be able to use the built-in zip utility by right clicking on the .txt file and selecting Compress (zip on Windows), as pictured in Figure 18.

This should create a .zip file that you can upload to the web application.
Step 3
On the web application, use the “Choose Files” button under “Upload Training Data” and select the zip file from your local file system.

Step 4
Additionally, from the dropdown underneath Train Data Type, select the ”Zipfile of URLs” option to tell the application that you are passing in a zipfile of URLs.

Step 5
Additionally, the user will need to specify the type of model to be trained by the application. In order to specify the type of model to be returned by the application, select the appropriate model type from the ”Model Type” dropdown.

Step 6
To start training a model, select the Perform Training button to send the data through the application. The application will then return a model that can be downloaded using the Download button.

![Perform Training](image)

Figure 21: Perform Training

**Step 7**
After training has completed, the application will display the loss curve of the model during the training process. A good loss curve indicating that the model learned the task well should show decreasing loss (y-axis) as training time increases (x-axis increases). A sample loss curve is shown in [Image 16](image).

### 7.2.3 Text Data

**Step 1**
It may be simplest for the user to upload a folder of .txt files, each containing a document related to the crisis event of interest to be used in the training of the model. In this case, the user should upload a .zip file containing multiple .txt files, where each .txt file is a specific document related to the crisis event the model is being trained on. To create a .txt file, use a text editor such as TextEdit to create a .txt file. Note that the application does not currently support .docx or .pages files, but that support will hopefully be added in the future. For each web article used to train the model, copy the entirety of the relevant content of a particular web page and paste the text into a created .txt file.

**Step 2**
Once the .txt files containing the contents of each web page have been created, then use a zip utility to create a zipfile of the .txt files. Alternatively, on Windows or MacOS, you may be able to use the built-in zip utility by right clicking on all of the .txt files and selecting Compress (zip on Windows), as pictured in Figure [22](image).

This should create a .zip file that you can upload to the web application.

**Step 3**
On the web application, use the “Choose Files” button under “Upload Training Data” and select the zip file from your local file system.

**Step 4**
Additionally, from the dropdown underneath Inference Data Type, select the “Zipfile of .txt files” option to tell the application that you are passing in a zipfile of .txt files.

**Step 5**
Additionally, the user will need to specify the type of model to be trained by the application. In order to specify the type of model to be returned by the application, select the appropriate model type from the “Model Type” dropdown.
Figure 22: Compress .txt files

Figure 23: Upload a zip file of text files

Figure 24: Specify the type of model to train
Step 6 To start training a model, select the “Perform Training” button to send the data through the application. The application will then return a model that can be downloaded using the “Download” button.

![Perform Training](image)

Figure 25: Perform Training

Step 7 After training has completed, the application will display the loss curve of the model during the training process. A good loss curve indicating that the model learned the task well should show decreasing loss (y-axis) as training time increases (x-axis increases). A sample loss curve is shown in [16].

7.3 Selecting Model Threshold

Once a one-class classification model has been trained, it is necessary to determine what the optimal cosine similarity value to use as a threshold for classification is. As, for example, a one-class autoencoder outputs a high-dimensional vector representation of the original input, we need some way to determine whether the document is related to the specific class or not using this high-dimensional vector. One way to accomplish this is to take the cosine similarity of the output vector with respect to each of the preprocessed training documents, averaging these scores, and determining if the resulting scalar is above a certain threshold. All values above the threshold would then be classified as related, while all values above the threshold would be classified as not related to the crisis event as interest. However, this requires determining a specific threshold value for each training dataset and model type. To accomplish this task, our application uses an additional dataset to generate a threshold between the training and inference stages. This dataset can be uploaded in a variety of formats. Currently, the same formats of HTML web pages, URLs, and text data are supported.

7.3.1 HTML Web Pages

Step 1

If this is the preferred data format, users should download two sets of web pages, one containing web pages related to the crisis event of interest and one set containing web pages unrelated to the crisis event of interest. These two datasets will be used to generate a threshold to be used by the model during inference. In a web browser, right-click on the webpages to be used in labeling and click “Save As” to save the website as an HTML document, as seen in Figure [11]. Note that the related webpages should be different from your training webpages.

Step 2
Once you have saved all of your webpages, then use a zip utility to create a zipfile each set of saved HTML web pages. Alternatively, on Windows or MacOS, you may be able to use the built-in zip utility by right clicking on all of the .html files and selecting Compress (zip on Windows), as pictured in Figure 12. This should create two separate .zip files that you can upload to the web application.

**Step 3**

On the web application, use the “Choose Files” button under “Upload Related Labeling Data” and select the zip file of related web pages from your local file system.

**Step 4**

On the web application, use the “Choose Files” button under “Upload Unrelated Labeling Data” and select the zip file of unrelated web pages from your local file system.

**Step 5**

Additionally, from the dropdown underneath Data Type, select the “Zipfile of HTML” option to tell the application that you are passing in a zipfile of HTML web pages.

**Step 6**

Additionally, the user will need to specify the type of model that was trained by the application. In order to specify the type of model, select the appropriate model type from the “Model Type” dropdown.

**Step 7**

To start the labeling process, select the “Perform Labeling” button to send the data through the application.

**Step 8**

Once labeling is completed, three visualizations will appear. The first visualization will be a slider allowing the user to manually adjust the threshold, if desired. The application will by default select the cosine similarity threshold that maximizes the accuracy of the classification problem. This slider is shown in Figure 26.

![Figure 26: Labeling Threshold Slider](image)

**Step 9**

The second visualization is a plot of the average cosine similarity of each document to all training documents, colored based on whether the documents were related or not. A sample visualization is shown in Figure 27.

**Step 10**

The final visualization is a list of all uploaded documents along with information relevant to the specific document, such as the document name, the cosine similarity, the actual label of the document, and the label predicted by the application given the set cosine similarity threshold value. This visualization is shown in Figure 28.
Figure 27: Cosine Similarity Plot

Figure 28: Documents List
7.3.2 URLs

**Step 1** To create two datasets of URLs, users should go to the web pages of interest, some related to a specific crisis event and others unrelated for proper thresholding. The user should first compile a list of URLs to serve as related documents. This can be accomplished by creating a text document using a text editor such as TextEdit for Mac and opening a new .txt file. Note that Word (.docx) and Pages (.pages) files will not work for the application at the moment, we are working on adding support for these formats in the future.

Once the text file has been created, the user should add URLs to web pages related to the crisis event of interest to the document. To add a particular URL to the document, go to the web page, select the URL from the search bar, copy the URL from the search bar, and paste the URL in the text document, as seen in Figure 17.

The same steps should be taken to create a list of URLs unrelated to the crisis event of interest.

**Step 2** Once you have saved all of your URLs, then use a zip utility to create two zip files of the text files. Alternatively, on Windows or MacOS, you may be able to use the built-in zip utility by right clicking the text file and selecting Compress (zip on Windows), as pictured in Figure 18.

This should create two separate .zip files that you can upload to the web application.

**Step 3** On the web application, use the “Choose Files” button under “Upload Related Labeling Data” and select the zip file of related URLs from your local file system.

**Step 4** On the web application, use the “Choose Files” button under “Upload Unrelated Labeling Data” and select the zip file of unrelated URLs from your local file system.

**Step 5** Additionally, from the dropdown underneath Data Type, select the “Zipfile of URLs” option to tell the application that you are passing in a zip file of HTML web pages.

**Step 6** Additionally, the user will need to specify the type of model that was trained by the application. In order to specify the type of model, select the appropriate model type from the “Model Type” dropdown.

**Step 7** To start the labeling process, select the “Perform Labeling” button to send the data through the application.

**Step 8** Once labeling is completed, three visualizations will appear. The first visualization will be a slider allowing the user to manually adjust the threshold, if desired. The application will by default select the cosine similarity threshold that maximizes the accuracy of the classification problem. This slider is shown in Figure 26.

**Step 9** The second visualization is a plot of the average cosine similarity of each document to all training documents, colored based on whether the documents were related or not. A sample visualization is shown in Figure 27.

**Step 10** The final visualization is a list of all uploaded documents along with information relevant to the specific document, such as the document name, the cosine similarity, the actual label of the document, and the label predicted by the application given the set cosine similarity threshold value. This visualization is shown in Figure 28.
7.3.3 Text Files

Step 3 On the web application, use the “Choose Files” button under “Upload Related Labeling Data” and select the zip file of related text files from your local file system.

Step 4 On the web application, use the “Choose Files” button under “Upload Unrelated Labeling Data” and select the zip file of unrelated text files from your local file system.

Step 5 Additionally, from the dropdown underneath Data Type, select the “Zipfile of .txt files” option to tell the application that you are passing in a zipfile of text files.

Step 6 Additionally, the user will need to specify the type of model that was trained by the application. In order to specify the type of model, select the appropriate model type from the “Model Type” dropdown.

Step 7 To start the labeling process, select the “Perform Labeling” button to send the data through the application.

Step 8 Once labeling is completed, three visualizations will appear. The first visualization will be a slider allowing the user to manually adjust the threshold, if desired. The application will by default select the cosine similarity threshold that maximizes the accuracy of the classification problem. This slider is shown in figure 26.

Step 9

The second visualization is a plot of the average cosine similarity of each document to all training documents, colored based on whether the documents were related or not. A sample visualization is shown in figure 27.

Step 10 The final visualization is a list of all uploaded documents along with information relevant to the specific document, such as the document name, the cosine similarity, the actual label of the document, and the label predicted by the application given the set cosine similarity threshold value. This visualization is shown in figure 28.

7.4 Generating Predictions

Once a model has been successfully trained, the application can also be used to generate classification predictions from a set of sample inputs. This input data can be submitted in multiple formats, including as HTML, URLs, or text data.

7.4.1 HTML Web Pages

Step 1 To create a dataset of HTML web pages, users should go to the web pages of interest, some related to a specific crisis event and others unrelated for proper testing. In a web browser, right-click on the webpages of interest and click “Save As” to save the website as an HTML document, as seen in Figure 11.

Step 2 Once you have saved all of your webpages, then use a zip utility to create a zipfile of all of the saved HTML web pages. Alternatively, on Windows or MacOS, you may be able to use the built-in zip utility by right clicking on all of the .html files and selecting Compress (zip on Windows), as pictured in Figure 29.

Step 3 On the web application, use the “Choose Files” button under “Upload Inference Data” and select the zip file from your local file system.
**Step 4** On the web application, use the ”Choose Files” button under ”Upload Inference Model” and select the trained model you received as output from the application during the training step.

**Step 5** Additionally, from the dropdown underneath Inference Data Type, select the ”Zipfile of HTML” option to tell the application that you are passing in a zipfile of HTML web pages.

**Step 6** Additionally, the user will need to specify the type of model the application will use to generate data. Note that the model the user uploads should match the actual type of model selected, or predictions will not be generated correctly. In order to specify the type of model to be returned by the application, select the appropriate model type from the ”Model Type” dropdown. We are currently working on expanding the number of model types supported.

**Step 7** To start, select the ”Perform Inference” button to send the data through the application. The application will then use the model to generate predictions for the data that
was input to the application. The predictions can be downloaded using the “Download” button.

![Perform Inference](image)

**Figure 32: Perform Inference**

**Step 8** The visualization produced is a list of all uploaded documents along with information relevant to the specific document, such as the document name, the cosine similarity, the actual label of the document, and the label predicted by the application given the set cosine similarity threshold value. This visualization is shown in figure 33.

![Documents List](image)

**Figure 33: Documents List**

### 7.4.2 URLs

**Step 1** To create a dataset of URLs, users should go to the web pages of interest, some related to a specific crisis event and others unrelated for proper testing. The user should first compile a list of URLs to train the model on. This can be accomplished by creating a text document using a text editor such as TextEdit for Mac and opening a new .txt file. Note that Word (.docx) and Pages (.pages) files will not work for the application at the moment, we are working on adding support for these formats in the future.

Once the text file has been created, the user should add URLs to web pages related to the crisis event of interest to the document. To add a particular URL to the document, go to the web page, select the URL from the search bar, copy the URL from the search bar, and paste the URL in the text document, as seen in Figure 34.

![Copy the URL from the search bar](image)

**Figure 34: Copy the URL from the search bar**

**Step 2** Once you have saved all of your URLs, then use a zip utility to create a zipfile of the text file. Alternatively, on Windows or MacOS, you may be able to use the built-in zip utility by right clicking the text file and selecting Compress (zip on Windows), as pictured in Figure 35.

![Compress file](image)

**Figure 35: Compress file**
Step 3 On the web application, use the “Choose Files” button under “Upload Inference Data” and select the zip file from your local file system.

Step 4 On the web application, use the “Choose Files” button under “Upload Inference Model” and select the trained model you received as output from the application during the training step.

Step 5 Additionally, from the dropdown underneath Inference Data Type, select the “Zipfile of URLs” option to tell the application that you are passing in a zipfile of URLs.

Step 6 Additionally, the user will need to specify the type of model the application will use to generate data. Note that the model the user uploads should match the actual type of model selected, or predictions will not be generated correctly. In order to specify the type of model to be returned by the application, select the appropriate model type from the “Model Type” dropdown. We are currently working on expanding the number of model types supported.
Step 7 To start, select the “Perform Inference” button to send the data through the application. The application will then use the model to generate predictions for the data that was input to the application. The predictions can be downloaded using the “Download” button.

Figure 38: Perform Inference

Step 8 The visualization produced is a list of all uploaded documents along with information relevant to the specific document, such as the document name, the cosine similarity, the actual label of the document, and the label predicted by the application given the set cosine similarity threshold value. This visualization is shown in figure 33.

7.4.3 Text Data

Step 1 To create a dataset consisting of .txt files, the user should upload a .zip file containing multiple .txt files, where each .txt file is a specific document. Some of the documents should be related to the crisis event the model is being trained on, while others should be unrelated to allow for a good testing set. To create a .txt file, use a text editor such as TextEdit, as opposed to Word or Pages. Note that the application does not currently support .docx or .pages files, but that support will hopefully be added in the future. For each web article used to generate predictions from, copy the entirety of the relevant content of a particular web page and paste the text into a created .txt file.

Step 2 Once the .txt files containing the contents of each web page have been created, then use a zip utility to create a zipfile of the .txt files. Alternatively, on Windows or MacOS, you may be able to use the built-in zip utility by right clicking on all of the .txt files and selecting Compress (zip on Windows), as pictured in Figure 22.

This should create a .zip file that you can upload to the web application.

Step 3 On the web application, use the “Choose Files” button under “Upload Inference Data” and select the zip file from your local file system.

Step 4 On the web application, use the “Choose Files” button under “Upload Inference Model” and select the trained model you received as output from the application during the training step.

Step 5 Additionally, from the dropdown underneath Inference Data Type, select the “Zipfile of .txt files” option to tell the application that you are passing in a zipfile of text files.

Step 6 Additionally, the user will need to specify the type of model the application will use to generate data. Note that the model the user uploads should match the actual type of model selected, or predictions will not be generated correctly. In order to specify the type of model to be returned by the application, select the appropriate model type from
the “Model Type” dropdown. We are currently working on expanding the number of model types supported.

**Step 7** To start, select the “Perform Inference” button to send the data through the application. The application will then use the model to generate predictions for the data that was input to the application. The predictions can be downloaded using the “Download” button.

**Step 8** The visualization produced is a list of all uploaded documents along with information relevant to the specific document, such as the document name, the cosine similarity, the actual label of the document, and the label predicted by the application given the set cosine similarity threshold value. This visualization is shown in figure 33.
8 Developer Manual

8.1 Repository

All code is located in a GitLab repository, hosted on code.vt.edu, located at https://code.vt.edu/calebmcirvin111/cs4624oneclasscrisisevents. Once added as a contributor, this repository may be cloned using HTTPS by cloning https://code.vt.edu/calebmcirvin111/cs4624oneclasscrisisevents.git or via SSH at git@code.vt.edu:calebmcirvin111/cs4624oneclasscrisisevents.git.

8.1.1 Prerequisites

A number of packages and prerequisites are needed to run the application. Firstly, an installation of Python 3 is necessary. It is recommended to install Python 3 packages in a virtual environment to ensure consistency between developers, though this is not strictly necessary.

Flask needs to be installed. In your Python virtual environment, run

$ pip install flask flask_cors

There may be other packages that need to be installed as well, including tqdm, torch, gensim, and pandas. These may also be installed with pip in the same way.

$ pip install tqdm torch gensim pandas

Note that additional files, such as the gensim KeyedVectors file, were not included in the repository due to space considerations.

8.1.2 Installation

Once the repository has been cloned and required packages have been installed, enter the main folder structure, cd into the server folder, and start the Flask API server using the command

$ python app.py

To start the frontend Vue website, cd into the "cs_4624_occ_pipeline" folder and run

$ npm install

and

$ npm run dev
The server should start on something similar to `http://127.0.0.1:5173`. Navigate to this URL and the website should load, ready to be interacted with.

### 8.2 File Structure

The most important files in the repository are described below. Note that some files irrelevant to the application itself are omitted, as well as some files deemed unnecessary to describe (e.g., less impactful Vue components).

```
/  
  src......................................................... Stores content files for the frontend
  |  assets................................................. Stores .css files for markup
  |  components........................................... Stores different Vue components
  |  |  Home.vue............................................. Landing page
  |  |  Train.vue........................................... Training page
  |  |  Label.vue........................................... Label page
  |  |  Inference.vue....................................... Inference page
  |  |  router.................................................. Stores router
  |  |  |  index.js............................................. Routes functionality between pages
  |  |  App.vue................................................ Vue template
  |  |  main.js............................................... Creates application
  |  index.html.............................................. Home page HTML document
  |  package.json........................................... Stores a list of node packages
  |  server............................................... Stores backend, Flask server
  |  |  app.py................................................. Backend functionality for training the model
```

### 9 Data Files

Below are examples of the three different types of data the application supports: URLs, HTML, and text files.

#### 9.1 HTML

HyperText Markup Language, or HTML, is a format for structuring content on the Internet.

Below is a sample of an HTML document.

```
The input data should be a zipfile consisting of multiple HTML documents.
```

#### 9.2 URLs

The input data should be a single file consisting of multiple URLs, as seen in Figure 44.
9.3 Text files

The below is an example text file for uploading to the application.

10 Deliverables

Our deliverables consist primarily of the source code for the text classification pipeline, as well as development reports detailing the project for future clients / developers planning on interacting with the project.

10.1 Backend

• Scripts to process HTML web pages
• Scripts to load input URLs
• Scripts to vectorize input text data
• Scripts to train different types of models depending on client specifications
• Scripts to perform inference given a one-text classification model and input text

10.2 Frontend
• Frontend providing a place for the client to interact with
• Frontend should have characteristics as described in Section 4

10.3 Accessibility
• Report describing our project for future clients and developers to view to gain an understanding of the project

11 Lessons Learned

11.1 Training a one-class SVM
While we learned numerous lessons during the development process of this project, one of the primary takeaways was how to develop a one-class classification model. To be able to test our application and complete the development of the training section of the application, we needed to develop a Python backend script to train a one-class classification model. Carefully reading scikit-learn documentation and relevant literature allowed us to develop a script that successfully trained a one-class classification model.

11.2 Future Plans
Looking ahead, we’re aiming to make our text classification model for crisis events even more versatile and effective. Our vision is about creating a tool that can handle a wide range of crisis scenarios and adapt as required. Testing remains crucial as we expand the application’s capabilities. We see it as an ongoing process - each new model we provide functionality for will be thoroughly tested to ensure its predictions are reliable. The integration of additional machine learning models is a key part of our future plans. These models will complement our existing architecture, adding another layer of analysis and classification.

We’re also making sure our model can handle different types of data. Crisis-related information comes in various formats, including web content. Our model can now process a zip file containing HTML documents, allowing it to extract and classify information from multiple URLs. This not only broadens the data types our model can handle but also keeps us ahead in terms of emerging information sources.
12 Challenges

The primary challenge we faced was training a successful one-class text classification model. While the initial manual data set curation to upload to the application was straightforward, creating a script to train a one-class classification model proved difficult. In addition, we initially attempted to solve the problem similarly to how we would go about solving a traditional machine learning problem. However, as one-class text classification is different from the traditional multi-class classification paradigm, we needed to update our model training scripts to reflect this fact.

13 Timeline

Below is a table with an approximation of our timeline as we worked on the project.

<table>
<thead>
<tr>
<th>Date</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sep 1</td>
<td>Initial client meeting</td>
</tr>
<tr>
<td>Sep 15</td>
<td>Initial manual dataset collected</td>
</tr>
<tr>
<td>Sep 29</td>
<td>Initial progress made on frontend and backend</td>
</tr>
<tr>
<td>Oct 13</td>
<td>Initial client meeting</td>
</tr>
<tr>
<td>Oct 27</td>
<td>Simple classification model</td>
</tr>
<tr>
<td>Nov 10</td>
<td>Better performing classification mode</td>
</tr>
<tr>
<td>Nov 24</td>
<td>Finish pipeline, clean up code</td>
</tr>
<tr>
<td>Nov 30</td>
<td>Final touchups, deployment on Linux server</td>
</tr>
</tbody>
</table>

14 Acknowledgements

We would like to thank our client, Dr. Mohamed Farag, for meeting weekly with us and guiding the direction of our project.