Mapping Natural Disaster Locations

from

Social Media Content

PSDS 4900 Capstone Technical Report

Jason Douglas Arbogast

jason.arbogast@gmail.com

Table of Contents

Introduction

In the emerging hours of an unfolding crisis determining the geographic extent, affected population, and urgency of need for the affected individuals are key challenges for directing resources and aid workers in natural disaster and humanitarian crisis response efforts. Traditional print, television and radio media reports are limited by the geography accessible to reporters due to infrastructure damage and roads made impassible, lengthy publication cycles, and editorial choice in coverage. Social media allows individuals directly impacted by the event to rapidly connect to the broader world and report events as they unfold nearest to the source, or to amplify the plight of family and associates in the affected areas.

The wealth of social media data available allows analysts to present facts on the ground for decision makers when allocating resources and responses. However, the high volume and high velocity of incoming social media presently overwhelms analysts attempting to decipher the most informative posts from social media sources. Natural language processing and classification using a neural network present a method to classify posts based on whether or not they are informative to natural disaster events and allow analysts to narrow their focus to the relevant data.

Some social media platforms provide coordinate information on posts, however, generating coordinates is dependent on a poster's device or application settings, or reliant on the poster manually tagging a location. Even if the poster allows their location to be tagged, the content of the post may mention a location of a natural disaster event occuring different from their device or profile location, or mention several locations at once. Geoparsing, the process of identifying location names in text and determining coordinates of the locations, provides a way to extract the locational information from a social media post and prepare for further spatial analysis and visualization.

Data should be mapped in a meaningful and consistent way so that even users with minimal experience in Geographic Information Science can gain insight and make conclusions. At present communicating easily understood graphics to leaders requires access to a geographic information system and other specialized software. For rapid identification of locations in an unfolding crisis, an interactive web map provides multiple users access to the most current data and the ability to render informative graphics within a web browser with low cognitive load allowing the user to spend more energy drawing insightful conclusions.

Combining these three tasks, discovering information in social media posts informative for natural disaster events, identifying locations mentioned in the content of those posts, and rendering the resultant data in a web application allows analysts to focus on finding the locations and populations facing the greatest challenges and needs in a crisis.

Project Workflow Overview

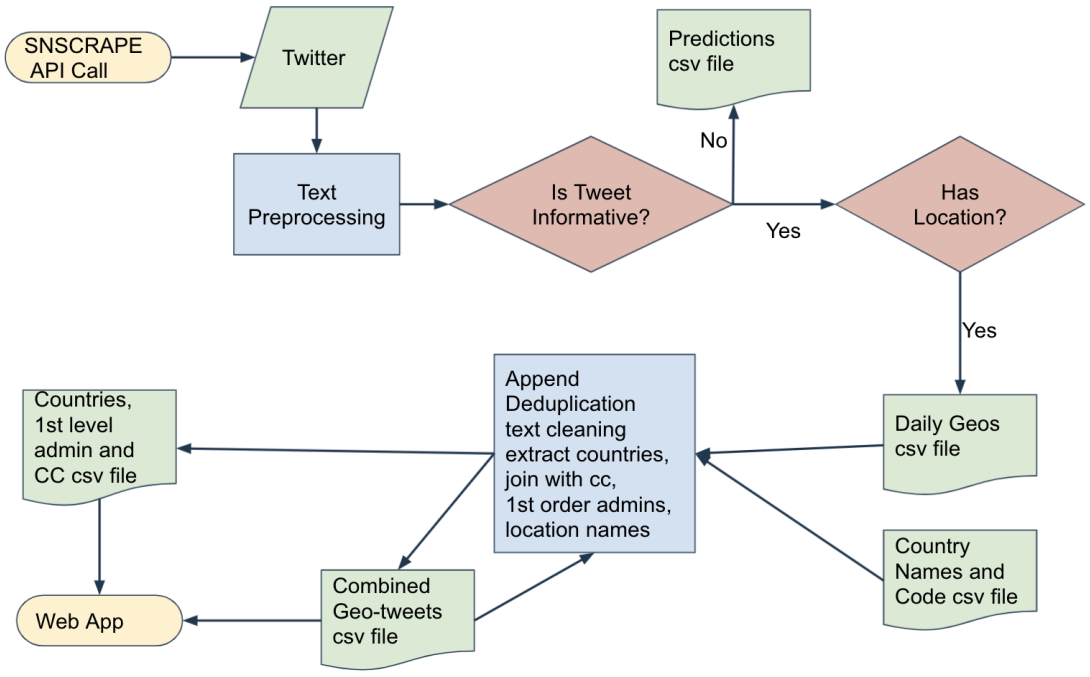

Figure 1: Flowchart of Project

This project explored a workflow and method to condition, classify, geolocate and map unstructured data from social media in order to discover clusters of locations mentioned in

natural disaster social media posts. This method collected posts from Twitter leveraging the scscrape python library, and processed and conditioned the text of the posts. A binary classification recurrent neural network was trained in TensorFlow with Keras using human labeled tweets aggregated by the CrisisBenchmark dataset created by the Crisis NLP project at the Qatar Computing Research Institute. The text of the training data was vectorized using a pre-trained text embedding built from tweets using the Global Vectors for Word Representation (GloVE) methodology and available from the GloVe Project at Stanford University (Pennington, Socher, and Manning 2014). Hyperparameters for the model were tuned with the Hyperband algorithm, and the final model was evaluated using a 5-fold cross-validation. The model and text embedding built from the training data were used to classify tweets pulled from June and July 2021 for natural disaster informativeness. The resulting informative tweets were then parsed for location and georeferenced using the Mordecai python library. Finally, the informative posts with locations mentioned in the text were displayed in an interactive R shiny web application for end users to map location explore by filtering data by geography, date and time, and research interest; discover trending topics on their selected data in a word cloud and export data for use by analysts in further research and visualization.

Social Media Data Source Ingesting

This project used the snscrape python library to pass a query to Twitter's API and execute a search (Git repository: https://github.com/JustAnotherArchivist/snscrape) to pull tweets. The Twitter API object, returned as a JSON, included metadata about the tweet such as date and time of the post, language, and how many times the post has been liked, retweeted, or commented on. Snscrape conditioned and flattened the JSON as a python readable dictionary. The Twitter API also returned a Twitter geo object, depending on whether or not a user opts in to manually geotag a post, or their settings allow an exact GPS coordinates (Twitter, n.d.). A few practical problems arise with the use of the geo object from twitter which make the locational information of lesser utility. A user could post about events that occur in a disparate location

from where their device is located, rendering the GPS coordinates even if enabled irrelevant or even post about events at several occasions in one post, yet not tag all or any of the places leading to incomplete data. Twitter also passed tagged places as a bounding box, which would require identifying the center point of the polygon to convert to a point representation. Finally twitter users geotag posts at rates between 0.85% (Sloan et al. 2013) to 2.31% (Huang and Carley 2019, 3). As such this study will not use the Twitter geo object to determine the relevant location but will leverage a geoparser to identify text based locational information.

```
def twittsearch(text_query,since_date,until_date,tweetcount):

    tweets_list = []

    query = '{0} since:{1} until:{2} filter:has_engagement'.format(text_query, since_date, until_date)
    print(query)

    # Using TwitterSearchScraper to scrape data and append tweets to list

    for i,tweet in enumerate(sntwitter.TwitterSearchScraper(query).get_items()):

        if i>tweetcount:
            break
        tweets_list.append([tweet.date, tweet.id, tweet.content, tweet.url,
                            tweet.lang,tweet.retweetedTweet,tweet.quotedTweet])
    # Creating a dataframe from the tweets list above
    tweets_df = pd.DataFrame(tweets_list,
                             columns=['Datetime', 'TweetId', 'Text','TweetLink',
                                      "Language", "RTFrom", "QTFrom"])

    print("found {} tweets ranging from {} to {}"
          .format(len(tweets_df),
                  tweets_df.Datetime.min(),tweets_df.Datetime.max()))

    print("dropping duplicates")
    tweets_df = tweets_df.drop_duplicates(subset=['TweetId'])
    print("total of tweets now: {}".format(len(tweets_df)))
    print("english only")

    tweets_df = tweets_df[tweets_df["Language"]=='en']
    print("total of tweets now: {}".format(len(tweets_df)))
    tweets_df = tweets_df[tweets_df["RTFrom"].isna()]
    print("total of tweets now: {}".format(len(tweets_df)))
    tweets_df = tweets_df[tweets_df["QTFrom"].isna()]
    print("total of tweets now: {} ranging from {} to {}".
          format(len(tweets_df),
                 tweets_df.Datetime.min(),tweets_df.Datetime.max()))

    tweets_df = tweets_df[['Datetime', 'TweetId', 'Text','TweetLink']]
    return tweets_df

scrapestart =  datetime.datetime.now()
```

Figure 2: Twitter search function

```
print("Scraping commenced at {}".format(scrapestart))
# ============================================================================
text_query = '("forest fire") OR wildfire OR bushfire OR \
(extreme heat) OR (record heat) OR heatwave OR ("heat wave") OR typhoon OR cyclone OR hurricane OR \
tornado OR ("storm surge") OR  blizzard OR snow OR ("ice storm") OR sleet OR thunderstorm OR \
hail OR flood OR flooding OR freeze OR frost OR (extreme cold) OR landslide OR tsunami OR ("tidal wave") OR \
earthquake OR eruption OR volcano OR lava OR lahar OR avalanche OR mudslide OR sinkhole'

since_date = '2021-06-01'
until_date = '2021-06-02'
tweetcount = 200000
twts = twittsearch(text_query,since_date,until_date,tweetcount)

# ============================================================================
scrapend =  datetime.datetime.now()
print("Scraping ended at {}".format(scrapend))
print("Scraping time {}".format(scrapend-scrapestart))
```

Figure 3: Example search parameters

Given the limitations of hardware and time, this study leveraged the text query capability of snscrape by sending an inclusive boolean query of common disaster terms to the Twitter API, and filter for only tweets with engagement (such as a like, retweet, or comment). Once collected, duplicate tweet IDs were dropped, any tweet which was itself either a retweet or quote tweet was dropped and only english language tweets were retained. Daily pulls for data were conducted from 1 June 2021 to 31 July 2021. A total of over 3 million tweets were collected. Each day averaged about 52,000 tweets. The maximum tweets pulled for a single day of about 109,764 occurred on 11 July while the minimum occurred on 20 June with 26,140 tweets. Even though tweets were filtered with a boolean query of common natural disaster terms within the collected data irrelevant tweets still existed. One key word used in the query was “lightning”. Two spikes between 7 July and 11 July 2021, as seen in Figure 4, were likely due Twitter users posting about the Tampa Bay Lightning winning the Stanley Cup Final and subsequent parade. While this may have been a disaster for hockey fans in Montreal, these tweets were not relevant for the purpose of mapping natural disaster locations. This underscored the need for additional classification to help reduce the irrelevant tweets in the data.

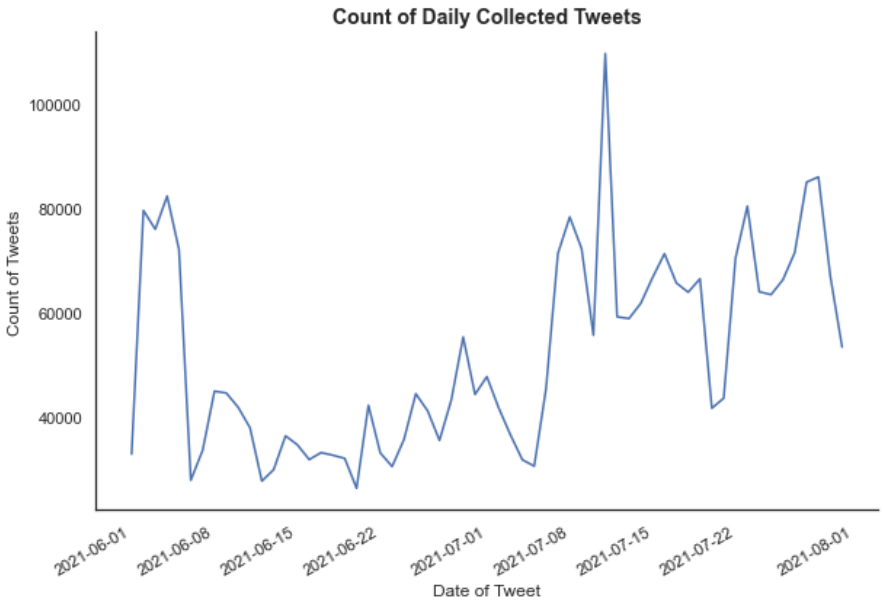

Figure 4: Daily count of tweets

_____On average the hour of day with the least number of tweets of collected English language tweets is about 0700 UTC (0300 US ET/ 0000 US PT) while the maximum is observed between 1600 and 2100 UTC (1400 US ET/1100US PT) and (1800 US ET/1500US PT).

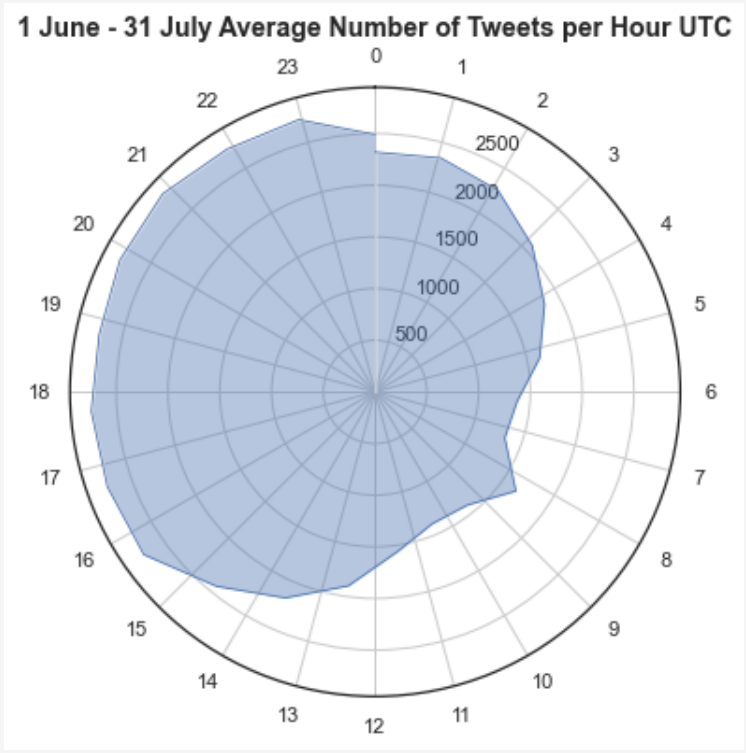

Figure 5: Tweets by time of day

Selection and exploration of Training Data

Training data were acquired from the CrisisBenchmark dataset created by the Crisis NLP project at the Qatar Computing Research Institute. Human volunteers had previously annotated several disaster datasets of tweets and this data consolidated those posts into one overarching dataset in order to facilitate research in crisis informatics and compare results of training models (Alam, F 2021). Each of the contributory datasets were manually labeled and validated. Further as the data were aggregated identical tweets from disparate datasets were removed. The final data was split into two larger groups: crisis_consolidated_informativeness_filtered_lang, “Informativeness”, and crisis_consolidated_humanitarian_filtered_lang, “Humanitarian”. The Informativeness dataset consisted of 156,452 English language tweets labeled as two classes: “informative” or “not informative”. The Humanitarian dataset consisted of 132,544 English language tweets labeled as 16 classes: “not_humanitarian", "other_relevant_information", "donation_and_volunteering", "requests_or_needs", "sympathy_and_support", "Infrastructure_and_utilities_damage", " Affected_individual", " caution_and_advice", "injured_or_dead_people", "disease_related", "response_efforts", "Personal_update", "Missing_and_found_people", "displaced_and_evacuations", "Physical_landslide", or "terrorism_related”.

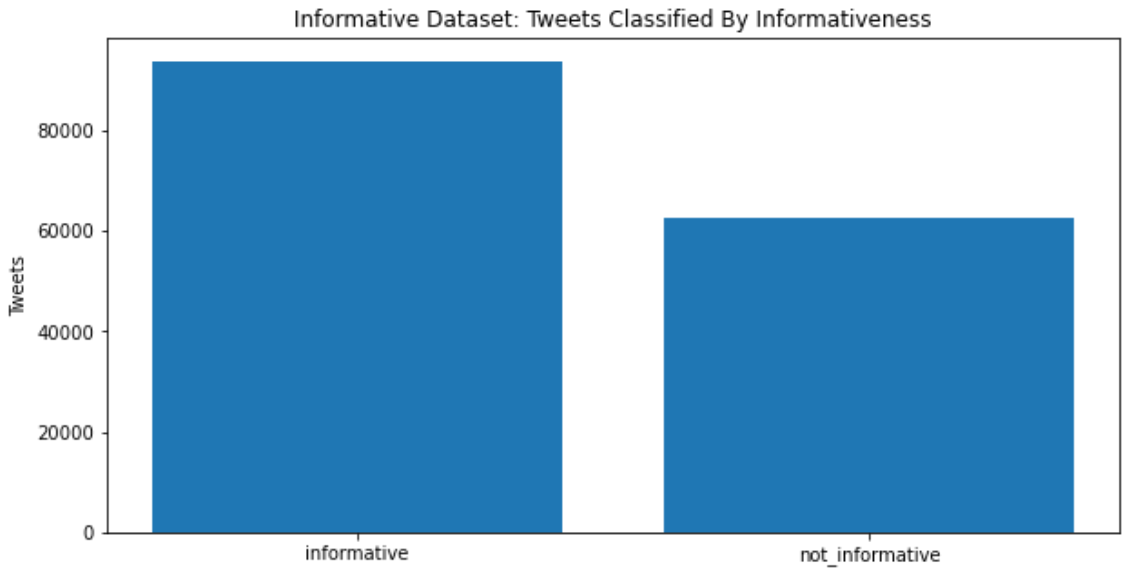

Figure 6: Informativeness Dataset Distribution

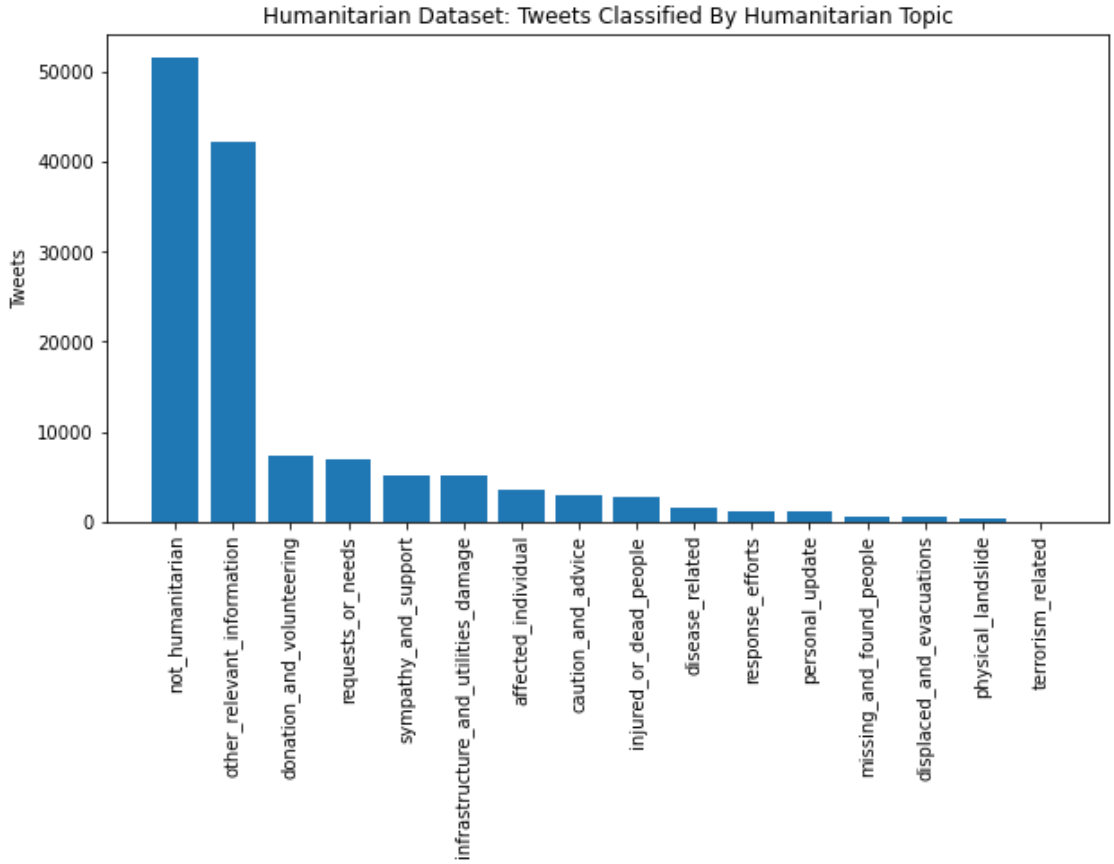

Figure 7: Humanitarian Dataset Distribution

Both datasets capture tweets from 61 disaster events from 2011 to 2017. Most of the events consist of fewer than 5,000 tweets, but come from a variety of disaster types and from many locations throughout the world. The varied locations of origin and events should help more reliably classify tweets from around the world as opposed to training data built on just US events.

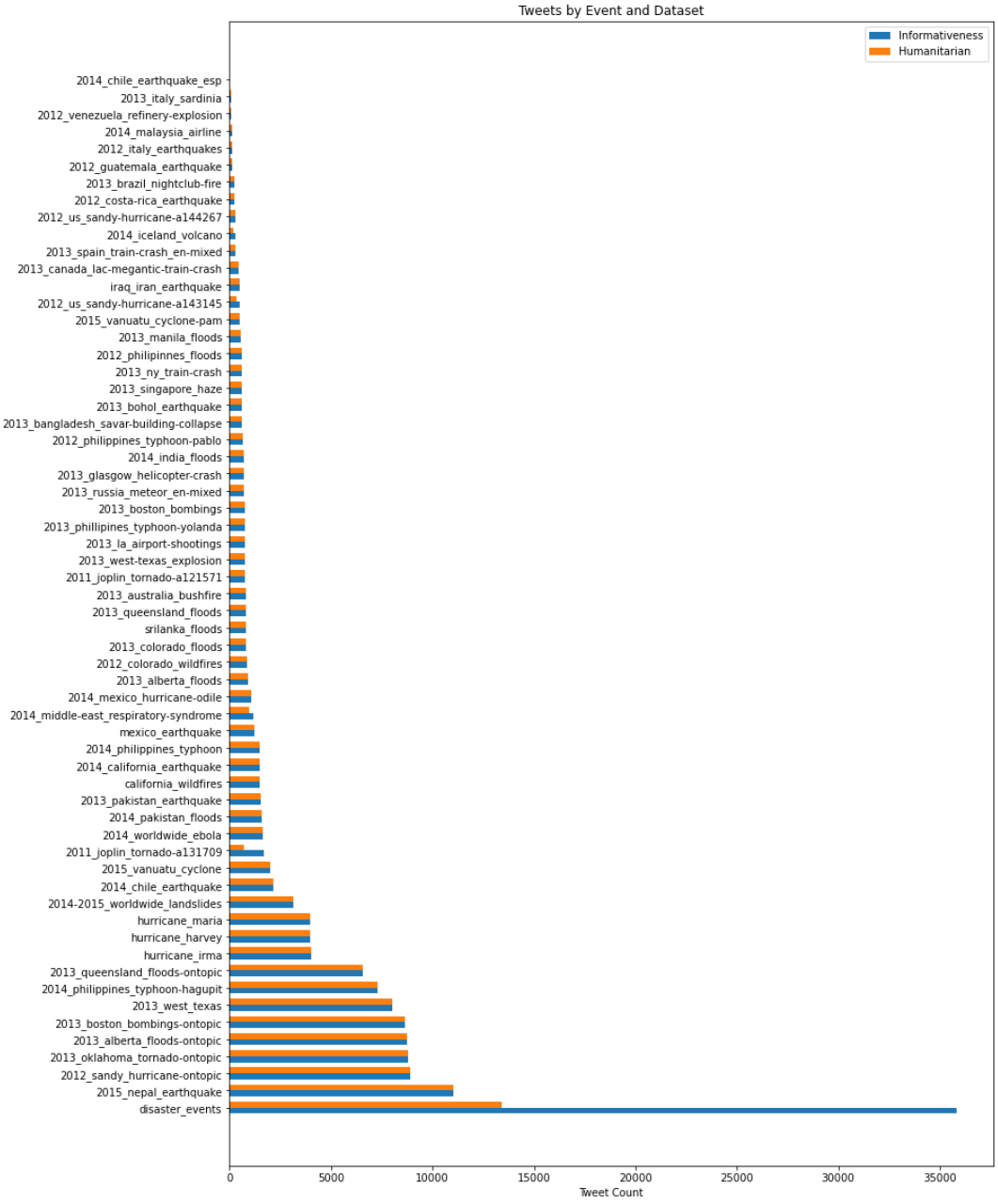

Figure 8: Composition of Training Data by Event

The Informativeness dataset was lightly cleansed and split into training and test data sets. The string values for the class labels column were converted to numerical values; two .csv files were created by a stratified random sample to ensure that the class labels were proportional in each dataset and to use the same split during repeated model testing and tuning.

```
informative_df.class_label =  informative_df.class_label.astype('category')
informative_df.class_label
informative_df["class_label_cat"] = informative_df.class_label.cat.codes
informative_df = informative_df.groupby(by=['class_label_cat'])
test = informative_df.sample(frac = 0.25, random_state = 42)
test.to_csv(os.path.join(parent_dir,"Data","InformativenessTest_Processed.csv)")
informative_df = filtered[filtered["lang"] == 'en']
train = informative_df.drop(test.axes[0])
train.to_csv(os.path.join(parent_dir,"Data","InformativenessTrain_Processed.csv)")
```

Figure 9: Train/Test split

Text conditioning and tokenization

The training data, test data, and scraped tweets had the same cleaning and tokenization steps applied to each dataset. The text of each tweet had the user name removed, links removed, punctuation removed, and numeral removed. The only difference was the text variable was overwritten in the train and test data while in the scraped data the cleaned text was written to the “ptext” variable so the original “text” was variable retained for geotagging.

```
# cleaning the text

def clean_text(text):
    '''Make text lowercase, remove links,remove punctuation
    and remove words containing numerals.'''
    text = text.lower()
    #get rid of usernames
    tweet_words = text.strip('\r').split(' ')
    for word in [word for word in tweet_words if '@' in word]:

            text = text.replace(word, "")
    #get rid of the re-tweet
    tweet_words = text.strip('\r').split(' ')
    for word in [word for word in tweet_words if 'rt' == word]:

            text = text.replace(word, "")

    text = re.sub('https?://\S+|www\.\S+', '', text)
    text = re.sub('[%s]' % re.escape(punctuation), '', text)
    text = re.sub('\n', '', text)
    text = re.sub('\w*\d\w*', '', text)
    return text
def remove_stopwords(text):
    words = [w for w in text if w not in stopwords.words('english')]
    return words
def lemmatize_text(text):
    lemmatizer = WordNetLemmatizer()
    return [lemmatizer.lemmatize(w) for w in text]  ##Notice the use of text.
def concatenate_text(text):
    return ' '.join(text)
```

Figure 10: Text Conditioning Helper Functions Used for Training and Scraped Data

```
# Applying the cleaning function to both test, train, and scraped datasets
train_data['text'] = train_data['text'].apply(lambda x: clean_text(x))
test_data['text'] = test_data['text'].apply(lambda x: clean_text(x))
train_data['text'] = train_data['text'].apply(lambda x:word_tokenize(x))
test_data['text'] = test_data['text'].apply(lambda x:word_tokenize(x))
train_data['text'] = train_data['text'].apply(lambda x : remove_stopwords(x))
test_data['text'] = test_data['text'].apply(lambda x : remove_stopwords(x))
train_data['text'] = train_data['text'].apply(lambda x : lemmatize_text(x))
test_data['text'] = test_data['text'].apply(lambda x : lemmatize_text(x))
train_data['text'] = train_data['text'].apply(lambda x : concatenate_text(x))
test_data['text'] = test_data['text'].apply(lambda x : concatenate_text(x))
twts['ptext'] = twts['Text'].apply(lambda x: clean_text(x))
twts['ptext'] = twts['ptext'].apply(lambda x: word_tokenize(x))
twts['ptext'] = twts['ptext'].apply(lambda x : remove_stopwords(x))
twts['ptext'] = twts['ptext'].apply(lambda x : lemmatize_text(x))
twts['ptext'] = twts['ptext'].apply(lambda x : concatenate_text(x))
```

Figure 11: Lambda functions to clean, tokenize, and lemmatize text

Binary Classification Model Construction and Training

Convolutional neural networks (CNN) have been shown to be more accurate, and have a more balanced recall to precision ratio than logistic regression or support vector machine techniques for binary classification (Nguyen et al. 2017, 6). Recurrent neural networks (RNN) are neural networks that allow previous outputs to be used as inputs, which allows for the sequence of data to be taken into account for learning. Since the next word in a sentence or tweet is dependent on previous words, and there is also meaning in the order of words this project chose to implement and RNN in prediction.

Text Vectorization

The RNN model was built with the python Keras and TensorFlow libraries. This allowed for a convenient way to test and visualize hyperparameters, and integrate text vectorizing. In order to prepare the text for machine learning, the values had to be converted to a numeric representation. Primary methods to establish the numerical representation of text values include Term Frequency-Inverse Document Frequency (TF-IDF) vectorization, initializing a vector from the training data or leveraging a pre-trained word embedding vector. A benefit of TF-IDF vectorization is that it can help highlight words which occur relatively infrequently in a collection of documents, but occur at a high rate in an individual document. However a major drawback is that using a TF-IDF vectorization creates a sparse matrix of text vectors, and converting that sparse matrix to a dense matrix requires a large amount of memory when building an RNN in

TensorFlow. Initializing a text vectorization from the training data itself had the benefit that building the vectorization integrated smoothly with the Keras model builder. However, this constructed the relationships between words exclusively on the training data and when new data were presented to the model the relationship did not hold and led to an overtrained model. To overcome this problem this project used a pre-trained word embedding vector to address the issue of novel data embedding. The Global Vectors for Word Representation (GloVe, https://nlp.stanford.edu/projects/glove/) pre-trained twitter vector is based on 2 billion tweets, 27 billion tokens, and has a 1.2 million word vocabulary stored in a 200 dimension vectorization (Pennington, Socher, and Manning 2014).

The training data set was split into training and validation sample and label lists so that the text could be converted to a TensorDataset and read by TensorFlow. The Glove file was loaded as a python dictionary, embedding index, with 1.2 million words each as a key for a 200 dimensional array. The training sample list was then mapped to the embedding index to create an embedding matrix and vectorizer for use in the neural network.

```
def train_val_split(df, validation_split):
    """
    This function generates the training and validation splits from an input dataframe

    Parameters:
        dataframe: pandas dataframe with columns "text" and "target" (binary)
        validation_split: should be between 0.0 and 1.0 and represent the proportion of the dataset to
        include in the validation split

    Returns:
        train_samples: list of strings in the training dataset
        val_samples: list of strings in the validation dataset
        train_labels: list of labels (0 or 1) in the training dataset
        val_labels: list of labels (0 or 1) in the validation dataset
    """

    text = df['text'].values.tolist()                          # input text as list
    targets = df['class_label_cat'].values.tolist()                    # targets

#   Preparing the training/validation datasets

    seed = random.randint(1,50)   # random integer in a range (1, 50)
    rng = np.random.RandomState(seed)
    rng.shuffle(text)
    rng = np.random.RandomState(seed)
    rng.shuffle(targets)

    num_validation_samples = int(validation_split * len(text))

    train_samples = text[:-num_validation_samples]
    val_samples = text[-num_validation_samples:]
    train_labels = targets[:-num_validation_samples]
    val_labels = targets[-num_validation_samples:]

    print(f"Total size of the dataset: {df.shape[0]}.")
    print(f"Training dataset: {len(train_samples)}.")
    print(f"Validation dataset: {len(val_samples)}.")

    return train_samples, val_samples, train_labels, val_labels
train_samples, val_samples, train_labels, val_labels = train_val_split(train_data, 0.25)
```

Figure 12: Training and Validation Split

```
path_to_glove_file = '/content/drive/MyDrive/Capstone/WordVector/WordVector/glove.twitter.27B.200d.txt'
embeddings_index = {}
f = open(path_to_glove_file, 'r', encoding='utf8')
for line in f:
    splitLine = line.split(' ')
    word = splitLine[0]                                    # the first entry is the word
    coefs = np.asarray(splitLine[1:], dtype='float32')     # these are the vectors representing word embeddings
    embeddings_index[word] = coefs
print("Glove data loaded! In total:",len(embeddings_index)," words.")
```

```
Glove data loaded! In total: 1193515  words.
```

Figure 13: Creating Embedding Index

```
def make_embedding_matrix(train_samples, val_samples, embeddings_index):
    """
    This function computes the embedding matrix that will be used in the embedding layer

    Parameters:
        train_samples: list of strings in the training dataset
        val_samples: list of strings in the validation dataset
        embeddings_index: Python dictionary with word embeddings

    Returns:
        embedding_matrix: embedding matrix with the dimensions (num_tokens, embedding_dim),
        where num_tokens is the vocabulary of the input data,
        and emdebbing_dim is the number of components in the GloVe vectors (can be 50,100,200,300)
        vectorizer: TextVectorization layer
    """

    vectorizer = TextVectorization(max_tokens=55000, output_sequence_length=50)
    text_ds = tf.data.Dataset.from_tensor_slices(train_samples).batch(128)
    vectorizer.adapt(text_ds)

    voc = vectorizer.get_vocabulary()
    word_index = dict(zip(voc, range(len(voc))))

    num_tokens = len(voc)

    hits = 0
    misses = 0

#   creating an embedding matrix
    embedding_dim = len(embeddings_index['the'])
    embedding_matrix = np.zeros((num_tokens, embedding_dim))
    for word, i in word_index.items():
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            # Words not found in embedding index will be all-zeros
            embedding_matrix[i] = embedding_vector
            hits += 1
        else:
            misses += 1

#     print("Converted %d words (%d misses)" % (hits, misses))
    print(f"Converted {hits} words ({misses} misses).")

    return embedding_matrix, vectorizer
```

Figure 14: Creating Embedding Matrix and Vectorizer

Hyperparameter Tuning

This project sought to strike a balance in the complexity of hyperparameters or values of the the number layers--number of individual functions in the RNN; the number of nodes--the number of weights of each of layers; when and where to apply dropout; which activation functions to use; and finally what learning rate to apply. A key trade off was learning time versus accuracy. The more complex a model, with more configurations to test the longer hyperparameter tuning took. Another factor considered was model overfitting during training. Overfitting was observed when training accuracy convergence is at a significantly higher level than validation accuracy. This model addressed overfitting through using a pre-trained text vectorization and using dropout between the layers of the network. Dropout cuts a random selection of the weights in a layer at the given rate, so that a dropout of 0.50, will cut half of the weights. This prevented the neural network from becoming over reliant on particular weights of a given node, and allows for better generalization as the drop is random.

To determine the optimal values of these model hyperparameters, the Hyperband function in the keras tuner was chosen. The hyperband function tested a few iterations of each configuration of the model, and discarded low performing configuration and let the highest performing configurations continue until convergence is reached by the validation loss ceasing to decrease. This saved significant time in tuning hyperparameters as opposed to a grid search which tests every possible configuration even those with substantially high validation loss values, and lower validation accuracy.

```
def myparamalam(hp):
    """
    This function initializes Keras model for binary text classification

    Parameters:
        embedding matrix with the dimensions (num_tokens, embedding_dim),
         where num_tokens is the vocabulary size of the input data,
          and emdebbing_dim is the number of components in the GloVe vectors

    Returns:
        model: Keras model
    """
    hp_num_nodes = hp.Choice('num_units', values =[16,32,64,128])
    hp_activation = hp.Choice('activator',values=['tanh','sigmoid', 'relu'])
    hp_activation2 = hp.Choice('activator2',values=['tanh','sigmoid', 'relu'])
    HP_DROPOUT_IN = hp.Choice('dropout_in_rate', values=[0.7, 0.8, 0.9])
    HP_DROPOUT = hp.Choice('dropout_rate', values=[0.4, 0.5, 0.6])
    hp_learning_rate = hp.Choice('learning_rate', values=[1e-2,1e-3,1e-4])

    num_tokens = embedding_matrix.shape[0]
    embedding_dim = embedding_matrix.shape[1]

    embedding_layer = Embedding(
    num_tokens,
    embedding_dim,
    embeddings_initializer=keras.initializers.Constant(embedding_matrix),
    trainable=False,                # we are not going to train the embedding vectors
    )


#   Here we define the architecture of the Keras model.
    int_sequences_input = keras.Input(shape=(None,), dtype="int64")
    x = embedding_layer(int_sequences_input)
    x = layers.Dropout(HP_DROPOUT_IN)(x)
    x = layers.Bidirectional(layers.LSTM(hp_num_nodes,
                                         dropout=HP_DROPOUT,
                                         return_sequences=True))(x)
    x = layers.Bidirectional(layers.LSTM(hp_num_nodes,
                                         dropout=HP_DROPOUT))(x)
    x = layers.Dense(hp_num_nodes,activation= hp_activation)(x)
    x = layers.Dropout(HP_DROPOUT)(x)
    preds = layers.Dense(1, activation=hp_activation2)(x)
    model = keras.Model(int_sequences_input, preds)

    # print('')
    # print("Training the model...")

    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=hp_learning_rate),
                loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
                metrics=['accuracy','binary_accuracy',])
    return model
```

Figure 15: Model Used for initial Hyperparameter Tuning

This project tuned hyperparameters in two steps. The first iteration held the node depth at 16, 32, 64, or 128 for all layers in a configuration, and held all dropouts other than the initial dropout at 0.4, 0.5, or 0.6 for configuration. Limiting the values in this way reduced the number of configurations at play, and was able to cull clearly underperforming candidate values. The optimal first dropout rate of 0.7, sigmoid was the best final activation function, and the optimal learning rate was 10^{-3}. No depth value clearly out performed any other on the first round, Only the values of 0.4, 0.5 were retested for dropout in the second round.

```
Search space summary
Default search space size: 7
num_units (Choice)
{'default': 16, 'conditions': [], 'values': [16, 32, 64, 128], 'ordered': True}
num_units2 (Choice)
{'default': 16, 'conditions': [], 'values': [16, 32, 64, 128], 'ordered': True}
num_units3 (Choice)
{'default': 16, 'conditions': [], 'values': [16, 32, 64, 128], 'ordered': True}
activator (Choice)
{'default': 'tanh', 'conditions': [], 'values': ['tanh', 'sigmoid', 'relu'], 'ordered': False}
dropout_rate (Choice)
{'default': 0.4, 'conditions': [], 'values': [0.4, 0.5], 'ordered': True}
dropout_rate2 (Choice)
{'default': 0.4, 'conditions': [], 'values': [0.4, 0.5], 'ordered': True}
dropout_rate3 (Choice)
{'default': 0.4, 'conditions': [], 'values': [0.4, 0.5], 'ordered': True}
```

Figure 16: Values Tested in Second Hyperparameter Tuning

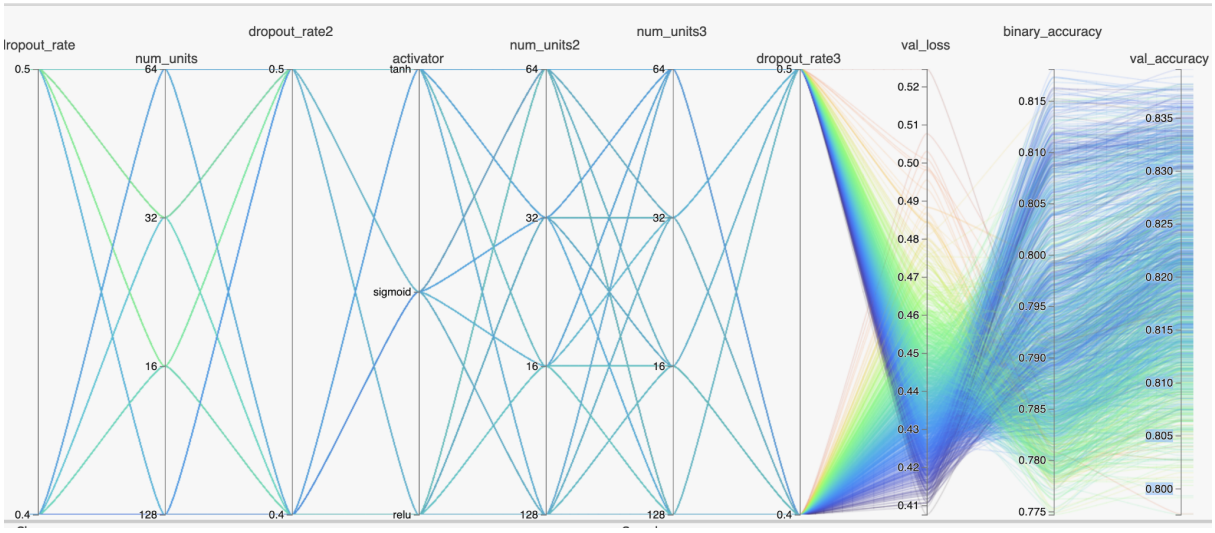

Figure 17: Parallel Plot of Values Tested in Second Hyperparameter Tuning

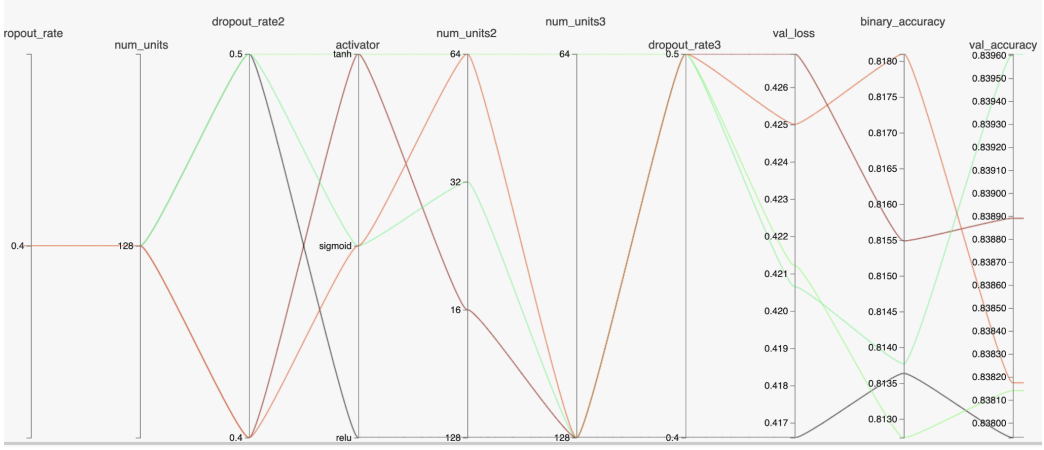

Figure 18: Using HiPlot to interactively view relationship of Hyperparameter to accuracy

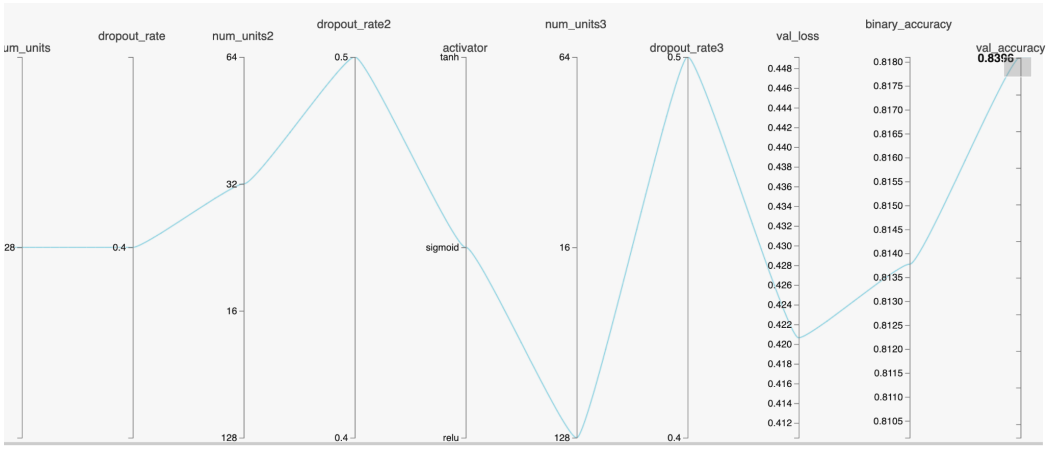

Figure 19: Wining Hyperparameter Configuration

Show 10 entries Search:

	uid	from_uid	num_units	num_units2	num_units3	activator	dropout_rate	dropout_rate2	dropout_rate3	val_loss	binary_accuracy	val_accuracy
■	977		128	32	128	sigmoid	0.4	0.5	0.5	0.42064911127090454	0.8137719631195068	0.8396059274673462
■	981		128	16	128	tanh	0.4	0.4	0.5	0.4268932640552521	0.8154877424240112	0.838890016078949
■	440		128	32	128	tanh	0.4	0.4	0.4	0.44666802883148193	0.8101471662521362	0.8381741046905518
■	500		128	64	128	sigmoid	0.4	0.4	0.5	0.42500585317611694	0.8181012272834778	0.8381741046905518
■	982		128	64	64	tanh	0.4	0.5	0.5	0.42121627926826477	0.8127379417419434	0.838140070438385
■	494		128	64	128	tanh	0.4	0.5	0.4	0.4491477608680725	0.8147491812705994	0.8380036950111389
■	974		128	128	128	relu	0.4	0.5	0.4	0.41660076379776	0.8136355876922607	0.8379355072975159
■	990		128	16	128	sigmoid	0.4	0.4	0.5	0.43687543272972107	0.8144082427024841	0.8378673195838928
■	496		128	128	64	tanh	0.4	0.5	0.4	0.4206501543521881	0.8143855333328247	0.8377309441566467
■	986		128	16	128	sigmoid	0.4	0.4	0.5	0.42910075187683105	0.8145446181297302	0.8377309441566467

Showing 1 to 10 of 1,742 entries Previous 1 2 3 4 5 ... 175 Next

Figure 20: Wining Hyperparameter Configuration in a table

5-Fold Cross-validation

After the second round of tuning the final model was trained with the best values from hyperparameter configuration then tested using a 5-fold cross-validation over all the data, bringing in the test data which had been held out since the beginning. The final training structure was as follows, a text embedding layer of 200 dimensions, a dropout layer of 0.7, a bi-directional long-short term memory(LSTM) layer, with a depth of 128 nodes and 0.4 drop out, a second bi-directional LSTM layer with a depth of 32 nodes, and 0.5 drop out, a dense layer

with a depth of 128 nodes and sigmoid activation, a dropout out of 0.5, then a dense layer with 1 node and sigmoid activation to complete the binary classification.

```
x_train = vectorizer(np.array([[s] for s in train_samples])).numpy()
x_val = vectorizer(np.array([[s] for s in val_samples])).numpy()
x_test = vectorizer(np.array([[s] for s in test_samples])).numpy()
y_train = np.asarray(train_labels).astype('float32').reshape((-1,1))
y_val = np.asarray(val_labels).astype('float32').reshape((-1,1))
y_test = np.asarray(test_labels).astype('float32').reshape((-1,1))

num_folds = 5
acc_list = []
acc_per_fold = []
loss_list = []
loss_per_fold = []
pcc_list = []
pcc_per_fold = []
rcc_list = []
rcc_per_fold = []
TP_list = []
TP_per_fold = []
FN_list = []
FN_per_fold = []
FP_list = []
FP_per_fold = []
TN_list = []
TN_per_fold = []
inputs = np.concatenate((x_train,x_val, x_test), axis=0)
targets = np.concatenate((y_train,y_val, y_test), axis=0)
kfold = KFold(n_splits=num_folds, shuffle=True)
fold_no = 1
```

Figure 21: Helpers to hold metrics of cross-validation

```
for train, test in kfold.split(inputs, targets):

    num_tokens = embedding_matrix.shape[0]
    embedding_dim = embedding_matrix.shape[1]

    embedding_layer = Embedding(
    num_tokens,
    embedding_dim,
    embeddings_initializer=keras.initializers.Constant(embedding_matrix),
    trainable=False,                # we are not going to train the embedding vectors
    )

    #   Here we define the architecture of the Keras model.
    int_sequences_input = keras.Input(shape=(None,), dtype="int64")
    x = embedding_layer(int_sequences_input)
    x = layers.Dropout(.7)(x)
    x = layers.Bidirectional(layers.LSTM(128,
                                         dropout=.4,
                                         return_sequences=True))(x)
    x = layers.Bidirectional(layers.LSTM(32,
                                         dropout=.5))(x)
    x = layers.Dense(128,activation= 'sigmoid')(x)
    x = layers.Dropout(.5)(x)
    preds = layers.Dense(1, activation='sigmoid')(x
    model = keras.Model(int_sequences_input, preds)

    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=['binary_accuracy',tf.keras.metrics.Precision(name='precision'),tf.keras.metrics.Recall(name='rec
                       tf.keras.metrics.TruePositives(name='true_positives'),tf.keras.metrics.FalseNegatives(name='fals
                       tf.keras.metrics.FalsePositives(name='false_positives'),tf.keras.metrics.TrueNegatives(name='tru

    # Generate a print
    print('------------------------------------------------------------------------')
    print(f'Training for fold {fold_no} ...')

    # Fit data to model
    history = model.fit(inputs[train], targets[train],
              batch_size=128,
              epochs=120,
              callbacks=[tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3,
                                                          restore_best_weights = True,
                                                          verbose = 1, min_delta = .0002 ),
                         tf.keras.callbacks.ModelCheckpoint(filepath=cppath,
                                                save_weights_only=True,
                                                verbose=1)
                        ],
              verbose=1)
    acc = history.history['binary_accuracy']
    pcc = history.history['precision']
    rcc = history.history['recall']
    TP = history.history['true_positives']
    FN = history.history['false_negatives']
    FP = history.history['false_positives']
    TN = history.history['true_negatives']
    loss = history.history['loss']

    acc_list.append(acc)
    loss_list.append(loss)
    pcc_list.append(acc)
    rcc_list.append(acc)
    TP_list.append(TP)
    FN_list.append(FN)
    FP_list.append(FP)
    TN_list.append(TN)

    # Generate generalization metrics
    scores = model.evaluate(inputs[test], targets[test], verbose=0)
    print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {model.metrics_names[1]} of {scores[1]*1
    {model.metrics_names[2]} of {scores[2]*100}%; {model.metrics_names[3]} of {scores[3]*100}%;/n\
    {model.metrics_names[4]} of {scores[4]}; {model.metrics_names[5]} of {scores[5]}/n\
    {model.metrics_names[6]} of {scores[6]}; {model.metrics_names[7]} of {scores[7]}')
    acc_per_fold.append(scores[1] * 100)
    loss_per_fold.append(scores[0])
    pcc_per_fold.append(scores[2] *100)
    rcc_per_fold.append(scores[3] *100)
    TP_per_fold.append(scores[4])
    FN_per_fold.append(scores[5])
    FP_per_fold.append(scores[6])
    TN_per_fold.append(scores[7])

    # Increase fold number
    fold_no = fold_no + 1
```

Figure 22:cross-validation of RNN

The 5-fold cross-validation results show that the accuracy of the final model was very close to validation accuracy of the best configuration of the hyperparameter tuning. Further the accuracy of the results of each fold were consistent. The average accuracy of the folds was approximately 84%. The model also converged to a minimum loss within 50 epochs. These consistent results demonstrate the model was well trained and was not overfit to the training data.

Precision is the measure of the percentage of posts classified as informative that were informative, given by the formula:

$$precision = True\ Positives \div (True\ Positives + False\ Positives)$$

The model had an overall precision of approximately 85.5% which indicated that 85.5% of the informative posts identified by the model were actually informative, while 14.5% were incorrectly identified as informative. This indicates the model performed well at clearing much of the non-informative posts, making analysis and culling through data easier for analysts

Recall is the percentage of the actually informative posts that were correctly identified by the model, given by the formula:

$$recall = True\ Positives \div (True\ Positives + False\ Negative)$$

The model had an overall recall of approximately 75.1% which indicated that the model identified 75.1% of the total informative posts as informative, while in misclassified 24.9% of actually informative posts as not informative.

The F1 score, or harmonic mean of recall and precision can show performance with imbalance data. The informative and not informative classes were not exactly split evenly, the F score can provide another measure of model performance.

$$F1 = (2 * Precision * Recall) / (Precision + Recall)$$

The F1 score of approximately 79.1% was within 5% of overall accuracy and the recall and precision indicating the model was consistent in classifying both classes.

```
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.37374749779701233 - Accuracy: 83.96344184875488%
 -Precision: 82.6337993144989% - Recall: 76.149630546569 82%
 -F1: 79.25931936303787
 -True Positives: 9588.0 - False Negatives: 3003.0
 -False Positives: 2015.0 - True Negatives: 16685.0
------------------------------------------------------------------------
> Fold 2 - Loss: 0.37376636266708374 - Accuracy: 84.07849073410034%
 -Precision: 82.18395709991455% - Recall: 76.46727561950684%
 -F1: 79.22262173877941
 -True Positives: 9498.0 - False Negatives: 2923.0
 -False Positives: 2059.0 - True Negatives: 16811.0
------------------------------------------------------------------------
> Fold 3 - Loss: 0.3705865740776062 - Accuracy: 84.11633372306824%
 -Precision: 83.51687192916 87% - Recall: 75.1238226890564%
 -F1: 79.09832585454039
 -True Positives: 9404.0 - False Negatives: 3114.0
 -False Positives: 1856.0 - True Negatives: 16916.0
------------------------------------------------------------------------
> Fold 4 - Loss: 0.3783569931983948 - Accuracy: 83.879834413528 44%
 -Precision: 84.57418084144592% - Recall: 73.578196763992 31%
 -F1: 78.69392560926507
 -True Positives: 9315.0 - False Negatives: 3345.0
 -False Positives: 1699.0 - True Negatives: 16931.0
------------------------------------------------------------------------
> Fold 5 - Loss: 0.3645106852054596 - Accuracy: 84.442311525344 85%
 -Precision: 85.05074381828308% - Recall: 74.208217859268 19%
 -F1: 79.26039526916752
 -True Positives: 9302.0 - False Negatives: 3233.0
 -False Positives: 1635.0 - True Negatives: 17120.0
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 84.09608244895935 (+- 0.19232831175872042)
> Precision: 83.59191060066223 (+- 1.0953529369042625)
> Recall: 75.10542869567871 (+- 1.1029730319887225)
> Loss: 0.3721936225891113(+- 0.004571330409003778)
> True Positives: 9421.4 (+- 109.09005454210755)
> False Negatives: 3123.6 (+- 152.2256220220499)
> False Positives: 1852.8 (+- 167.2894497570005)
> True Negatives: 16892.6(+- 143.8674389846431)
------------------------------------------------------------------------
Sum of Confusion Matrix
> Total True Positives: 47107.0
> False Negatives: 15618.0
> False Positives: 9264.0
> True Negatives: 84463.0
> Precision: 0.8356601798797254
> Recall: 0.7510083698684735
------------------------------------------------------------------------
```

Figure 23:cross-validation Results

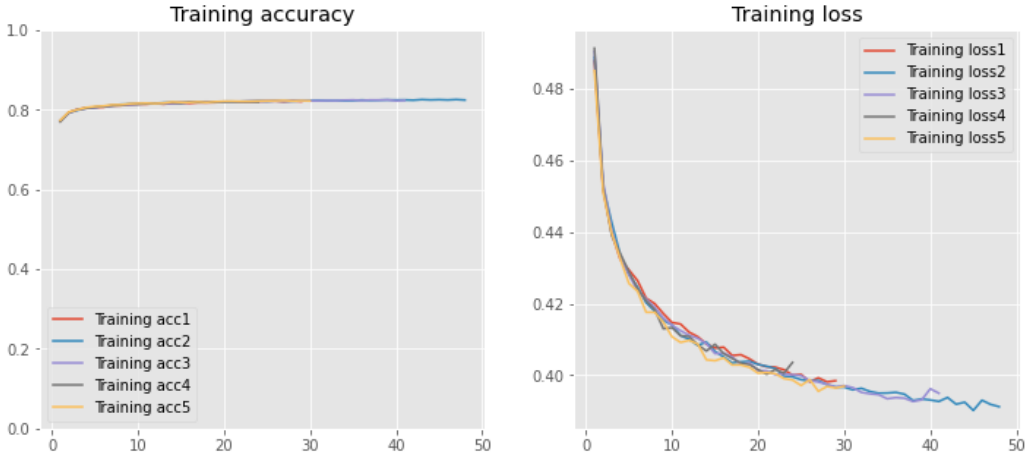

Figure 24: Accuracy and Loss Convergence across 5-Folds

```
Model: "model_18"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_19 (InputLayer)        [(None, None)]            0
_________________________________________________________________
embedding_18 (Embedding)     (None, None, 200)         11000000
_________________________________________________________________
dropout_36 (Dropout)         (None, None, 200)         0
_________________________________________________________________
bidirectional_36 (Bidirectio (None, None, 256)         336896
_________________________________________________________________
bidirectional_37 (Bidirectio (None, 64)                73984
_________________________________________________________________
dense_36 (Dense)             (None, 128)               8320
_________________________________________________________________
dropout_37 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_37 (Dense)             (None, 1)                 129
=================================================================
Total params: 11,419,329
Trainable params: 419,329
Non-trainable params: 11,000,000
_________________________________________________________________
```

Figure 25: Final Model Summary

This trained model was run against the scraped tweets to discover posts informative for natural disasters. The daily tweet counts of “informative” tweets compared to all tweets shows that, while there were still more posts classified as informative around 7 to 11 July, the classified data no longer demonstrated a dramatic spike as seen on the raw data over the same timeframe. This indicates the model may have effectively erased the influence of the Tampa Bay Lightning winning the Stanley Cup from data going forward.

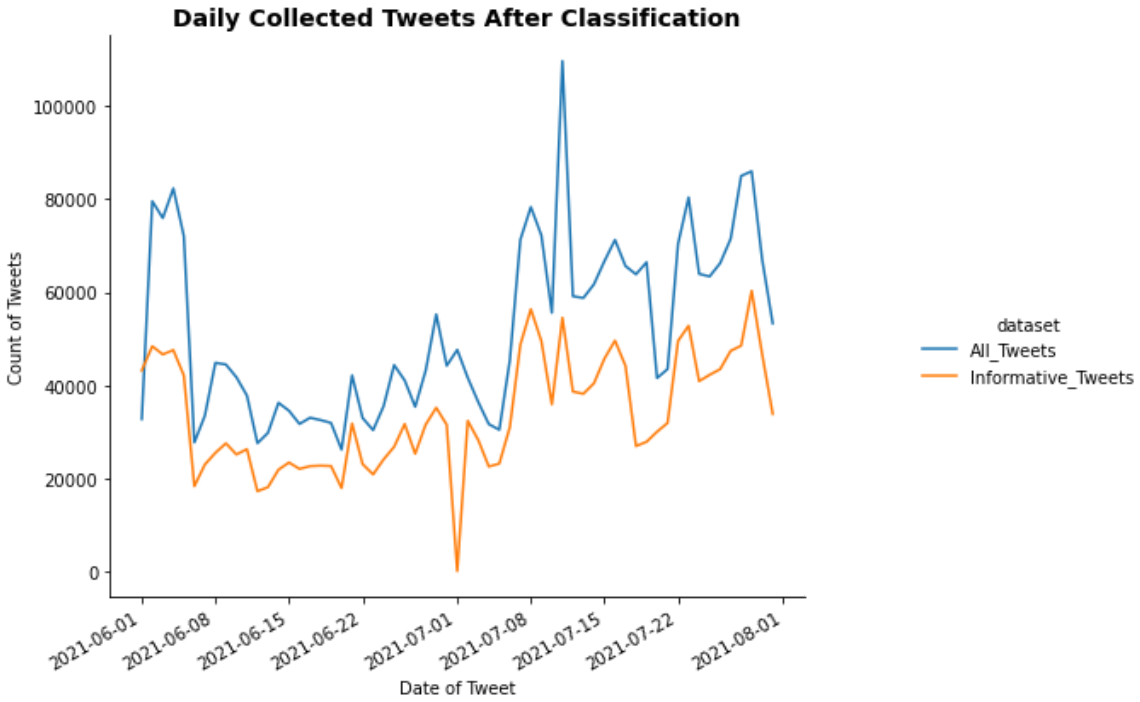

Figure 26: Effect of Classification Model on Scraped Tweets

Geoparsing Leveraging Mordecai

Mordecai uses the spaCy library's named entity recognition module to identify locations in a string. Mordecai then identifies candidate locations through querying a docker container running an Elasticsearch service with a Geonames index . In order to resolve the candidate locations, a neural network trained in keras is fed other context from the text such as the number of times the top two countries from the text are mentioned, the population of the country of the candidates. This should have countries mentioned more frequently in a document, and more populous countries receive greater weight. Given the limited context within a tweet to more marginal candidates this might lead to inaccurate locations being identified. Conversely there could also be fewer opportunities for the parser to take in noise about multiple locations in a large paragraph. The tweets classified as informative geoparsed using the Mordecai python library which mordecai returned as JSON string that was then flattened into a pandas dataframe then saved as a .csv file for mapping.

Building a Shiny Dashboard For Information Discovery

With the tweet scleaned, classified, and parsed, the final step in this project was to create an interactive web mapping application to explore clusters of natural disaster locations. The application is hosted and uses the app.R structure to hold both the UI and server code. Before the UI is rendered a few files are brought in to help populate drop down menus as well as the cleaned, predicted and geoparesed tweets. A few columns containing information from the mordecai geoparser were dropped, to help reduce file size and speed rendering. Finally a function to help convert the dataframe of filtered tweets to a geoJSON is passed so that it can be accessed later in the server function.With the Shiny Dashboard layout a sidebar panel allows the user to intuitively navigate the application lessenging cognitive load while navigating. From any point on the application a user can navigate to the about section, the map or the data table. Additionally the map data filters are held in the sidebar so the user can see what filters are applied to the data as they navigate between the pages.

```
countries <- read_csv("Countries - Countries.csv")
countrieslist <- countries[c('Country','alpha3')]
firstOrderAdmin <- read_csv('Admin_ones.csv')
tweets_geo <- read_csv('alltweet.csv',
                       col_select = -c(target,spans,country_predicted,country_conf,feature_class),
                       col_types = cols(TweetId = col_character()))

wgs84 = CRS("+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs")


spToGeoJSON <- function(x){
  # Return sp spatial object as geojson by writing a temporary file.
  # It seems the only way to convert sp objects to geojson is
  # to write a file with OGCGeoJSON driver and read the file back in.
  # The R process must be allowed to write and delete temporoary files.
  #tf<-tempfile('tmp',fileext = '.geojson')
  tf<-tempfile()
  writeOGR(x, tf,layer = "geojson", driver = "GeoJSON")
  js <- paste(readLines(tf), collapse=" ")
  file.remove(tf)
  return(js)
}
```

Figure 27: R code to load data on app load up

```
# Define UI for application using shiny dashboard
ui <- dashboardPage(
    dashboardHeader(title = "Natural Disaster Tweet Mapper",
                    titleWidth = 350),
    dashboardSidebar(
      width = 350,
        sidebarMenu(

            menuItem("About", tabName = "About", icon = icon("book")),

            menuItem("Map Mentioned Locations", tabName = "map", icon = icon("globe")),
            # This section will hold the data filters for the map and the table
            menuItem("Map Data Filters", icon = icon("filter"),
                     ## Time filter
                     fluidRow(column(width = 6,
                                     timeInput("startTime", "Start Time: ", value = NULL,
                                      seconds = FALSE, minute.steps = 15)),
                                     column(width = 6,
                                     timeInput("endTime", "End Time: ", value = NULL, seconds = FALSE,
                                               minute.steps = 15))),
                     ## Time filter
                     dateRangeInput( "dates", "Date Range: ",
                                     start = "2021-07-31", end = "2021-08-01",
                                     min = min(tweets_geo$Datetime), max = Sys.Date(),
                                     format = "yyyy-mm-dd", startview = "month",
                                     weekstart = 0,language = "en",
                                     separator = " to ",width = NULL, autoclose = TRUE),
```

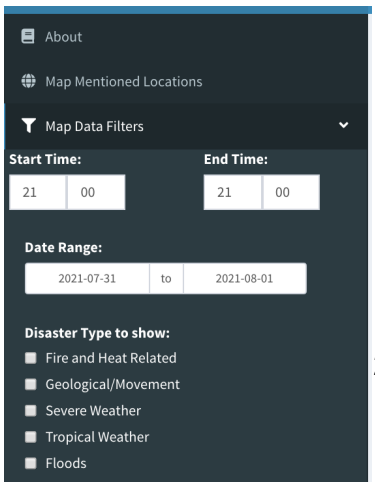

Figure 28: Shiny app UI code for sidebar and sidebar with filters for Data

The dashboard body displays an “about the app” screen on startup. Then by clicking the Map Mentioned Locations item in the sidebar a map is then displayed using leadleft to render a reactive map. The user can then filter data by date and time, the type of disaster they are interested in, and also drill the data to the country and first level administrative area. The map renders clusters at small scale, but as the user zooms the map renders locations as points, which the user can click to see information about the tweet. Finally a word cloud is rendered displaying the most frequently observed terms in the filtered tweets. The data table and export tab displays a data table rendered from the user defined filters and allows of export the filtered data.

```
    dashboardBody(
        tabItems(
            #About the app - create an html document to reference  rather than clutter up the shiny app
            tabItem(tabName = "About",includeHTML("about.html")),
            ##INFO BOXES AND MAP
            tabItem( tags$head(tags$style(HTML(
                    ".info {
                        padding: 6px 8px;
                        font: 14px/16px Arial, Helvetica, sans-serif;
                        background: white;
                        background: rgba(255,255,255,0.8);
                        box-shadow: 0 0 15px rgba(0,0,0,0.2);
                        border-radius: 5px;
                        width: 250px
                    }
                    .info h4 {
                        margin: 0 0 5px;
                        color: #777;
                    }"
                )),
                tags$link(rel = "stylesheet", type = "text/css",
                          href = "https://cdnjs.cloudflare.com/ajax/libs/leaflet/1.3.1/leaflet.css"),
                tags$link(rel = "stylesheet", type = "text/css",
                          href="https://unpkg.com/leaflet.markercluster@1.3.0/dist/MarkerCluster.css"),
                tags$link(rel = "stylesheet", type = "text/css",
                          href="https://unpkg.com/leaflet.markercluster@1.3.0/dist/MarkerCluster.Default.css")),
                tags$script(src="https://cdnjs.cloudflare.com/ajax/libs/leaflet/1.3.1/leaflet.js"),
                tags$script(src="https://unpkg.com/leaflet.markercluster@1.3.0/dist/leaflet.markercluster.js"),
              tabName = "map",
                column(width = 8,
                       box(width= 12,
                           title= strong(textOutput("MapTitle")),
                           tags$div(id="mapdiv", style="width: 100%; height: 700px;"),
                           tags$script(src="latest_map_leaflet.js"))

                ),
                column(width= 4,box(width= 12,

                    sliderInput("maxnumw","Maximum number of words:",min=1, max=100, step=5, value=25),
                    sliderInput("minfreqw","Minimum frequency:",min=1, max=500, step=10, value=200),),
                box(width= 12,plotOutput("wordc"))
               ),
               ),
            tabItem(tabName = "dataTable",downloadButton("download1"),dataTableOutput("mappedDataTable"))
            ) #tab items end
        ) #dasboard body end
)#ui end
```

Figure 29-a: Dashboard Body

Figure 29-b: About the App

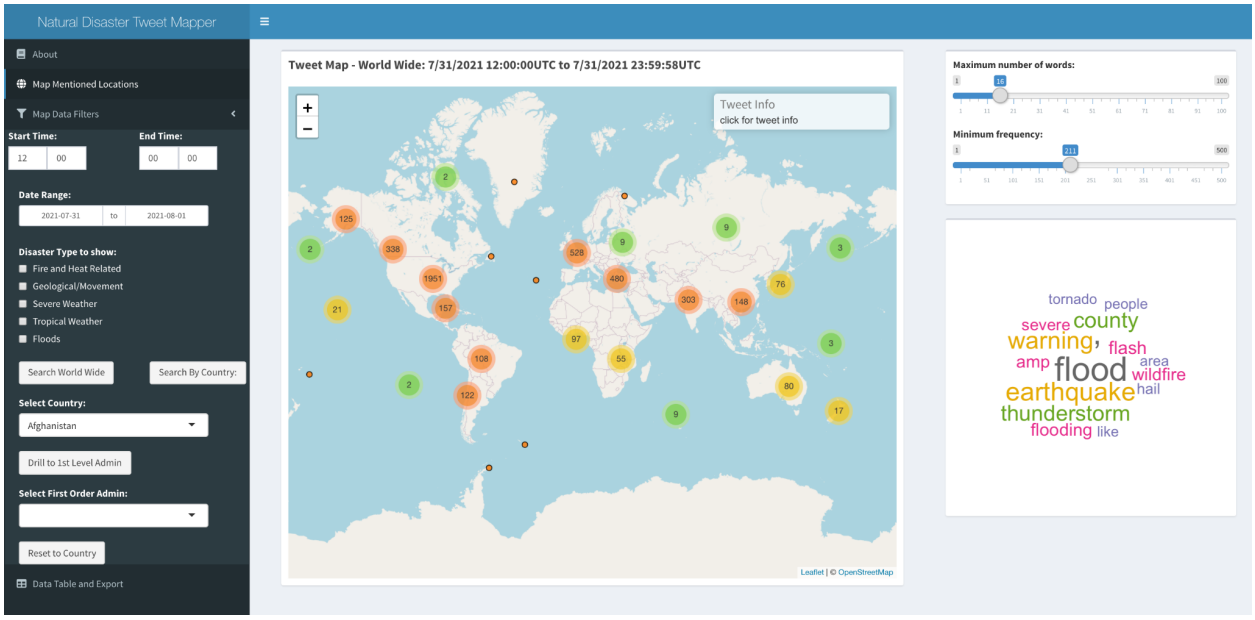

Figure 29-c: Map Pane

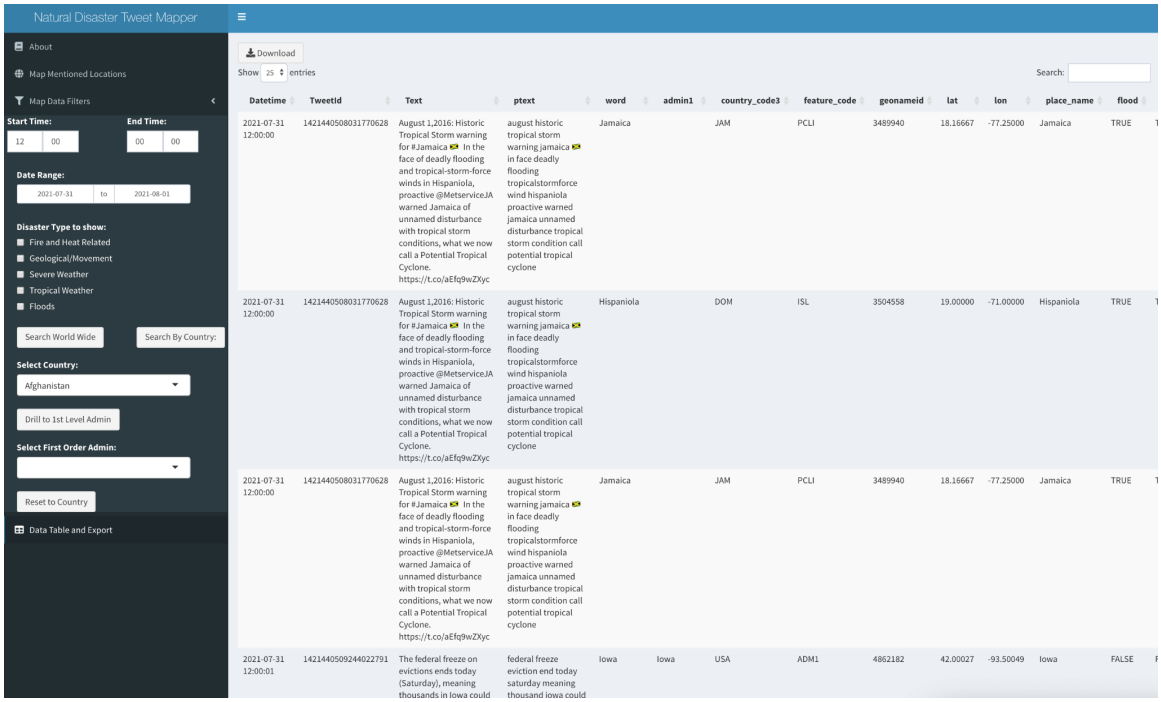

Figure 29-d: Data Pane

The server function activates once the user selects whether or not to filter by event and whether to search world wide or for a specific county. If the search world wide button is clicked; then tweets_geo dataframe is filtered by disaster types if a disaster had been checked, and date range, and converted to spatial points data frame then rendered as geoJSON to pass to leaflet. If the Search By country box is clicked, then tweets_geo dataframe is filtered by disaster types if a disaster was checked, date range, and country then converted to spatial points data frame then rendered as geoJSON to pass to leaflet. Based on which country is selected, the First Order Admin drop down is populated with that country's first order administrative divisions. If the user clicks the Drill to first level Admin button then the filters chosen at the country carry over and further refined to the selected first level admin area, and converted to spatial points data frame then rendered as geoJSON to pass to leaflet. The user can pop back to the national level by selecting the "Return to Country" button.

Following the selection the data frame is converted to a spatial points data frame, then converted to geoJSON, and sent as a message to leaflet to display on the map. A bounding box

is also passed to leaflet based on the extent of the spatial points data frame, with a 20 percent buffer, allowing the map to zoom to the selected geography of interest. Also the same data frame is sent to the Data Table and Export page. Lastly, a word cloud is rendered to help the user to easily get a sense of the key terms mentioned in their selected filters.

```
observeEvent(input$worldData,{
  print("World")
  listl<-length(input$DisasterType)
  if (listl == 0){tweets_geo_sub <-tweets_geo
  print('no filter')}
  else if (listl == 1){tweets_geo_sub <-subset(tweets_geo,
                                          ((tweets_geo[input$DisasterType[1]] == TRUE))) }
  else if(listl == 2){tweets_geo_sub<-subset(tweets_geo, ((tweets_geo[input$DisasterType[1]] == TRUE) |
                                                    (tweets_geo[input$DisasterType[2]] == TRUE)))}
  else if(listl == 3){tweets_geo_sub<-subset(tweets_geo,((tweets_geo[input$DisasterType[1]]  == TRUE |
                                                    (tweets_geo[input$DisasterType[2]] == TRUE) |
                                                    (tweets_geo[input$DisasterType[3]] == TRUE))))}
  else if(listl == 4){tweets_geo_sub<-subset(tweets_geo,((tweets_geo[input$DisasterType[1]]  == TRUE |
                                                    (tweets_geo[input$DisasterType[2]] == TRUE) |
                                                    (tweets_geo[input$DisasterType[3]] == TRUE) |
                                                    (tweets_geo[input$DisasterType[4]] == TRUE))))}
  else if(listl == 5){tweets_geo_sub<-subset(tweets_geo,((tweets_geo[input$DisasterType[1]]  == TRUE |
                                                    (tweets_geo[input$DisasterType[2]] == TRUE) |
                                                    (tweets_geo[input$DisasterType[3]] == TRUE) |
                                                    (tweets_geo[input$DisasterType[4]] == TRUE) |
                                                    (tweets_geo[input$DisasterType[5]] == TRUE))))}

  values <- reactiveValues(start = as.character(input$dates[1]),end= as.character(input$dates[2]))
  hrs <- reactiveValues(start = as.character(strftime(input$startTime,"%T")),
                        end = as.character(strftime(input$endTime,"%T")))


  tweets_geo_sub <- subset(tweets_geo_sub, tweets_geo_sub$Datetime >=
                                strptime(paste(values$start,hrs$start),format="%Y-%m-%d %H:%M:%S",tz="GMT") &
                                tweets_geo_sub$Datetime <
                                strptime(paste(values$end,hrs$end),format="%Y-%m-%d %H:%M:%S",tz="GMT"))

  output$MapTitle <-  renderText({ paste0("Tweet Map - World Wide: ",
                                          month(as.Date(min(tweets_geo_sub$Datetime))),"/",
                                          day(as.Date(min(tweets_geo_sub$Datetime))),"/",
                                          year(as.Date(min(tweets_geo_sub$Datetime))) ," ",
                                          str_sub(as.character(min(tweets_geo_sub$Datetime)),
                                                  start= -8),"UTC", ' to ',
                                          month(as.Date(max(tweets_geo_sub$Datetime))),"/",
                                          day(as.Date(max(tweets_geo_sub$Datetime))),"/",
                                          year(as.Date(max(tweets_geo_sub$Datetime))),' ',
                                          str_sub(as.character(max(tweets_geo_sub$Datetime)),
                                                  start= -8),"UTC"
  )})
```

Figure 30-a: Code to Render World Wide Search

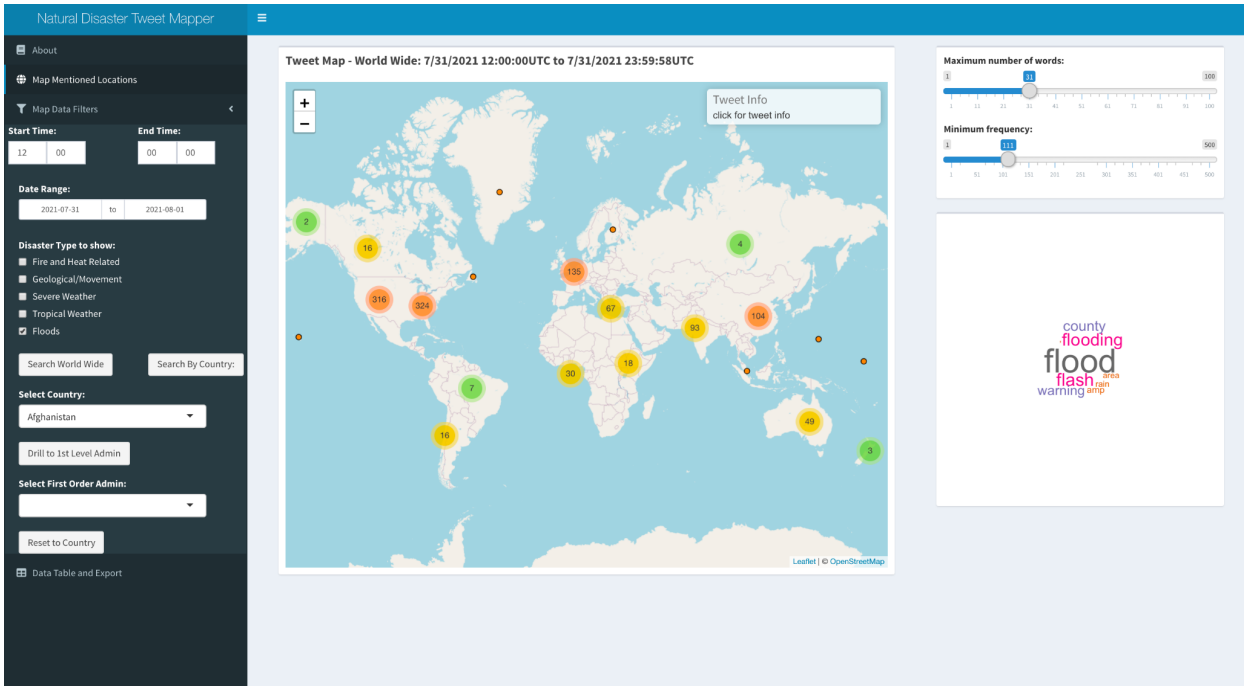

Figure 30-b: World Wide Search With Date and "Flood" Filter

```
observeEvent(input$CountrySearch, {

  selectedCountry<-isolate(input$CountrySelect)
  selectedNation<<-toString(countrieslist %>% filter(Country==selectedCountry)%>%select(alpha3))

  values <- reactiveValues(start = as.character(input$dates[1]),end= as.character(input$dates[2]))
  hrs <- reactiveValues(start = as.character(strftime(input$startTime,"%T")),
                        end = as.character(strftime(input$endTime,"%T")))

  listl<-length(input$DisasterType)
  if (listl == 0){tweets_geo_sub <-tweets_geo }
  else if (listl == 1){tweets_geo_sub <-subset(tweets_geo,
                                               ((tweets_geo[input$DisasterType[1]] == TRUE))) }
  else if(listl == 2){tweets_geo_sub<-subset(tweets_geo, ((tweets_geo[input$DisasterType[1]] == TRUE) |
                                                          (tweets_geo[input$DisasterType[2]] == TRUE)))}
  else if(listl == 3){tweets_geo_sub<-subset(tweets_geo,((tweets_geo[input$DisasterType[1]]  == TRUE |
                                                          (tweets_geo[input$DisasterType[2]] == TRUE) |
                                                          (tweets_geo[input$DisasterType[3]] == TRUE))))}
  else if(listl == 4){tweets_geo_sub<-subset(tweets_geo,((tweets_geo[input$DisasterType[1]]  == TRUE |
                                                          (tweets_geo[input$DisasterType[2]] == TRUE) |
                                                          (tweets_geo[input$DisasterType[3]] == TRUE) |
                                                          (tweets_geo[input$DisasterType[4]] == TRUE))))}
  else if(listl == 5){tweets_geo_sub<-subset(tweets_geo,((tweets_geo[input$DisasterType[1]]  == TRUE |
                                                          (tweets_geo[input$DisasterType[2]] == TRUE) |
                                                          (tweets_geo[input$DisasterType[3]] == TRUE) |
                                                          (tweets_geo[input$DisasterType[4]] == TRUE) |
                                                          (tweets_geo[input$DisasterType[5]] == TRUE))))}

  tweets_geo_sub <- subset(tweets_geo_sub, tweets_geo_sub$Datetime >=
                             strptime(paste(values$start,hrs$start),format="%Y-%m-%d %H:%M:%S",tz="GMT") &
                             tweets_geo_sub$Datetime <
                             strptime(paste(values$end,hrs$end),format="%Y-%m-%d %H:%M:%S",tz="GMT") &
                             tweets_geo_sub$country_code3 ==selectedNation)

  output$MapTitle <-  renderText({ paste0("Tweet Map - ",selectedCountry,": ",
                                    month(as.Date(min(tweets_geo_sub$Datetime))),"/",
                                    day(as.Date(min(tweets_geo_sub$Datetime))),"/",
                                    year(as.Date(min(tweets_geo_sub$Datetime))) ," ",
                                    str_sub(as.character(min(tweets_geo_sub$Datetime)),
                                            start= -8),"UTC", ' to ',
                                    month(as.Date(max(tweets_geo_sub$Datetime))),"/",
                                    day(as.Date(max(tweets_geo_sub$Datetime))),"/",
                                    year(as.Date(max(tweets_geo_sub$Datetime))),", ",
                                    str_sub(as.character(max(tweets_geo_sub$Datetime)),
                                            start= -8),"UTC"

  )})
  firstOrderAdminlist<<- firstOrderAdmin %>%
    filter(country_code3 == selectedNation) %>% select(admin1)
  updateSelectInput(session,"FirstOrderAdminSelect",
                    "Select First Order Admin: ",
                    choices = firstOrderAdminlist)
```

Figure 31-a: Code to Render Country Search

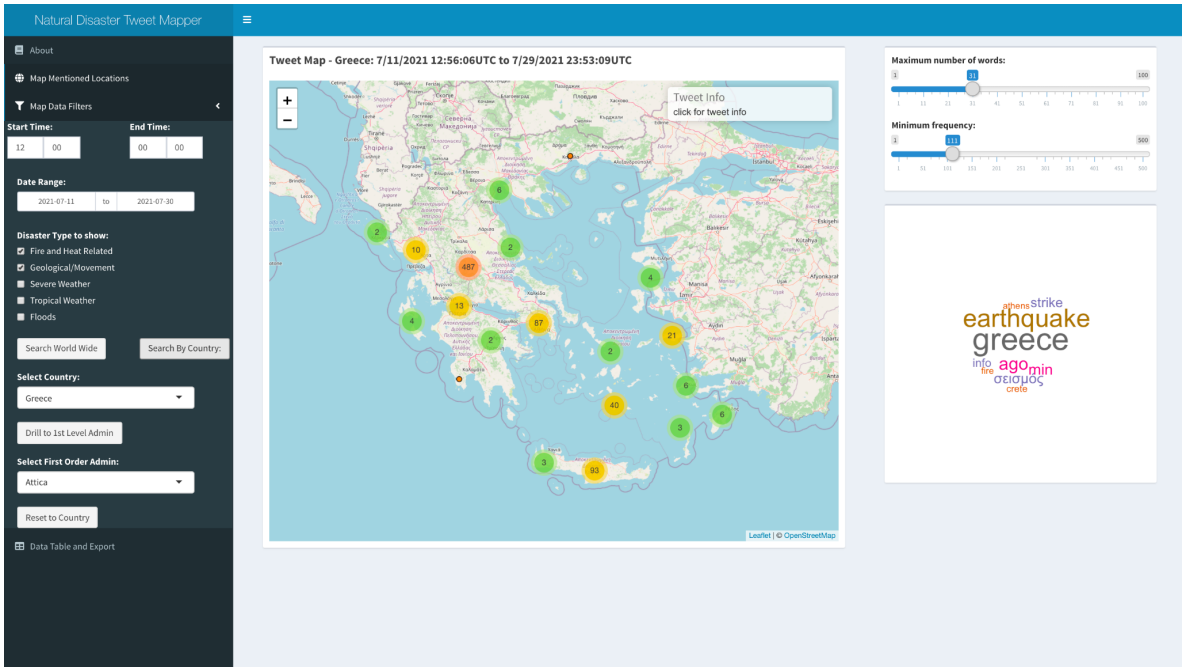

Figure 31-b: Filtered By County with 2 Disaster types Displayed

```
observeEvent(input$Drill,{
  values <- reactiveValues(start = as.character(input$dates[1]),end= as.character(input$dates[2]))
  hrs <- reactiveValues(start = as.character(strftime(input$startTime,"%T")),
                        end = as.character(strftime(input$endTime,"%T")))

  tweets_geo_sub <- subset(tweets_geo_sub, tweets_geo_sub$Datetime >=
                             strptime(paste(values$start,hrs$start),format="%Y-%m-%d %H:%M:%S",tz="GMT") &
                             tweets_geo_sub$Datetime <
                             strptime(paste(values$end,hrs$end),format="%Y-%m-%d %H:%M:%S",tz="GMT")&
                             tweets_geo_sub$country_code3 ==selectedNation &
                             tweets_geo_sub$admin1 ==input$FirstOrderAdminSelect)


  output$MapTitle <-  renderText({ paste0("Tweet Map - ",input$FirstOrderAdminSelect, " ",selectedCountry,": ",
                                          month(as.Date(min(tweets_geo_sub$Datetime))),"/",
                                          day(as.Date(min(tweets_geo_sub$Datetime))),"/",
                                          year(as.Date(min(tweets_geo_sub$Datetime))) ," ",
                                          str_sub(as.character(min(tweets_geo_sub$Datetime)),
                                                  start= -8),"UTC", ' to ',
                                          month(as.Date(max(tweets_geo_sub$Datetime))),"/",
                                          day(as.Date(max(tweets_geo_sub$Datetime))),"/",
                                          year(as.Date(max(tweets_geo_sub$Datetime))),"  ",
                                          str_sub(as.character(max(tweets_geo_sub$Datetime)),
                                                  start= -8),"UTC"
  )})
```

Figure 32-a: Code to drilled to First Level Admin

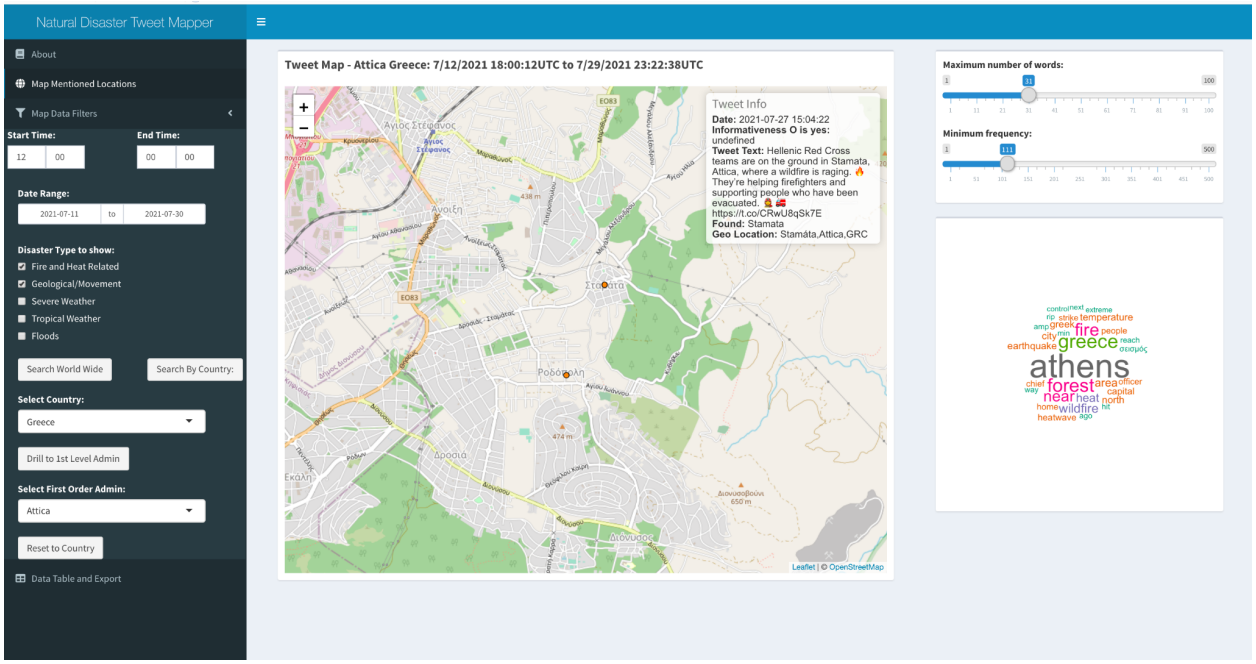

Figure 32-b: Map drilled to First Level Admin, displaying points and info box

```
  coords = data.frame(tweets_geo_sub$lon, tweets_geo_sub$lat)
  try({tweets_geo_spdf = SpatialPointsDataFrame(coords, tweets_geo_sub, proj4string = wgs84)})
  tweets_geoJSON<-fromJSON(spToGeoJSON(tweets_geo_spdf))
  output$mappedDataTable<-renderDataTable(tweets_geo_sub)

  output$download1 <- downloadHandler(
    filename = function() {
      paste0("Export.csv")
    },
    content = function(file) {
      write.csv(tweets_geo_sub, file)
    }
  )
  session$sendCustomMessage("load_map_geo_data", tweets_geoJSON)

  d1<-(bbox(tweets_geo_spdf)[4]-bbox(tweets_geo_spdf)[2])*.2
  d2<-(bbox(tweets_geo_spdf)[3]-bbox(tweets_geo_spdf)[1])*.2
  vector1 <- c(bbox(tweets_geo_spdf)[2]-d1,bbox(tweets_geo_spdf)[4]+d1)
  vector2 <- c(bbox(tweets_geo_spdf)[1]-d2,bbox(tweets_geo_spdf)[3]+d2)

  box<-array(c(vector1,vector2),dim = c(2,2))

  session$sendCustomMessage("bounds", box)
  output$wordc<- renderPlot ({
    lines<-tweets_geo_sub$ptext
    docs <- Corpus(VectorSource(lines))

    dtm2 <- TermDocumentMatrix(docs)
    m2 <- as.matrix(dtm2)
    v2 <- sort(rowSums(m2),decreasing=TRUE)
    d2 <- data.frame(word = names(v2),freq=v2)
    wordcloud(words = d2$word, freq = d2$freq, min.freq = input$minfreqw,
              max.words=input$maxnumw, random.order=FALSE, rot.per=0,
              colors=brewer.pal(8, "Dark2"))
  })

})
```

Figure 33: Map drilled to First Level Admin, displaying points and info box

Future Research and Conclusion

- An area of growth of this project would be to extend the methodology to a multi-class classification problem. The CrisisNLP project provides a 16 class dataset that could be used for that purpose. However, a challenge with a multi=class classification is that small group membership rates may lead to lower model accuracy and overfitting while training. Identifying neural network structures, and stratified sample methods that have done well on multiclass text classification, and testing to see how they perform on tweets would be the first step to accomplishing that goal.

The next key area would be to explore improving the speed and accuracy of geoparsing. The first step of the three step Mordecai geoparsing workflow spaCy leverages spaCy's named entity recognition module to identify candidate locations. Better integrating the text conditioning and word vectorization steps from the classification and geoparsing steps could increase overall model accuracy. A second approach would be to also build and update a new geonames gazetteer, or also fine tune a list of "stop locations" that have caused problems for the geoparser. For instance names of regions like the Midwest, New England and The South, refer to broad, non-discrete areas and while are extracted by Spacy, should probably be dropped from geoparsing in a project of this scope. Another area of interest would be using this method of classification and geoparsing to longer news articles. This may improve geoparsing, as mordecai also uses the context of a document to aid in classification. Conversely, there could be more noise and irrelevant information for locational classification. Additionally, identifying a single topic or informativeness of an article would be difficult. An area of research could be to use the NER to identify locations and discover the optimal amount of surrounding text for informativeness classification, and for getting the best geolocational accuracy.

Finally adopting a relational database, such as postgres to store and retrieve data could help bring the workflow to an enterprise level,and ensure data integrity and multiple user access. As of now pandas dataframes, and csv files are used to process and store data. While the shiny app loads and stores the data on the shiny.io server the original data could be corrupted, moved or deleted impeding updates. Building a relational database could store the data in an indexed, safe, and retrievable format.

This project successfully investigated a method to map locations mentioned in social media posts about natural disasters. A recurrent neural network for binary classification was trained using the Human labeled informativeness data set from the CrisisNLP protect. A pre-trained word vectorization from Glove built on tweets helped overcome model overfitting. Once the informative posts were identified the Mordecai python library was used to extract placenes, match to candidates using elasticsearch and geonames, and a pre-trained neural network classifier to geolocate the post. Finally the informative and mapped posts were deployed to a shiny web application. This allowed users to explore emerging locations, filter data according to their research geography and natural disaster tops, discover trending words in their filter, and finally export their findings for use in a geographic information system for more in depth analysis or to layer with other information sources. Future research will focus on improving geolocation parsing, extending workflow to multiclass classification, utilizing other media sources, and integrating long term data storage and a spatial RDBMS into the workflow. implemented a three stage process, using the Spacy named entity recognition module to find locations in the text, a geonames index on an elastic index to identify candidate matches, and scoring based context to feed it and it’s own binary geoparser glove.

Sources

Huang, Binxuan, and Kathleen Carley. 2019. “A Large-Scale Empirical Study of Geotagging Behavior on Twitter.” *In Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining,*, (August). http://www.casos.cs.cmu.edu/publications/papers/Twitter_geo_asonam_final.pdf.

Nguyen, Dat T., Kamela A. Al-Mannai, Shafiq Joty, Hassan Sajjad, Muhammad Imran, and Prasenjit Mitra. 2017. *Robust Classification of Crisis-Related Data on Social Networks using Convolutional Neural Networks*. Montreal, Quebec: Proceedings of the 11th International AAAI Conference on Web and Social Media (ICWSM). https://mimran.me/papers/deep_learning_for_crisis_data_classification_2016.pdf.

Pennington, Jeffery, Richard Socher, and Christopher D. Manning. 2014. “GloVe: Global Vectors for Word Representation.” https://nlp.stanford.edu/pubs/glove.pdf.

Sloan, Luke, Jeffery Morgan, William Housley, Matthew Williams, Adam Edwards, Pete Burnap, and Omer Rana. 2013. “Knowing the Tweeters: Deriving Sociologically Relevant Demographics from Twitter.” *Sociological Research Online* 18, no. 3 (August): 74-84. https://doi.org/10.5153/sro.3001.

Twitter. n.d. “Geo objects | Docs | Twitter Developer Platform.” Geo objects | Docs | Twitter Developer Platform. Accessed August 10, 2021. https://developer.twitter.com/en/docs/twitter-api/v1/data-dictionary/object-model/geo.

Links and Data Resources

Project Links

Project Github repository: https://github.com/arboj/arbogast-capstone

The code for the scraping, neural network, geoparsing, and web interface build is available on github. This allowed for version control of the project and for interested parties to recreate this project on their own systems and build upon this foundation

Shiny Web App: https://jarbo.shinyapps.io/tweetmap/

This shiny app is the web version of the shiny app that can be built from the code available on github. A key limitation is that shiny web hosting limits an instance to 8GB of ram, so loading excessive data can cause the app to crash.

Key Enabling Libraries and Programs

Snscrape https://github.com/JustAnotherArchivist/snscrape

Snscrape was used to query the twitter API and return tweets for analysis

Mordecai https://github.com/openeventdata/mordecai

Mordecai was used to geoparse locations from the content of tweets.

Docker https://www.docker.com

Docker was used it initiate a elastic search over a geonames index for the mordecai geo parser

Elastic https://www.elastic.co/guide/en/elasticsearch/reference/current/docker.html

Elastic was used to enable Mordecai to search the geonames index in a docker container

Data Sources

Training Data

CrisisBench: Benchmarking Crisis-related Social Media Datasets for Humanitarian Information Processing https://crisisnlp.qcri.org/crisis_datasets_benchmarks

Pre-trained Text Embedding

GloVe: Global Vectors for Word Representation https://nlp.stanford.edu/projects/glove/

Gazetter

Geonames https://www.geonames.org/