

FLORIDA INTERNATIONAL UNIVERSITY



Knight Foundation School of Computing and Information Sciences

Summer 2022 Senior Design Project

***Monitoring and mitigating misinformation that
disrupts adaptation to climate driven health threats***

Team member(s): Roberto Lopez-Trigo, Colleene McCurdy, Nathan Chiche, Brian Rodriguez, Kyle Ghani, Kyle Zimmer, Javier Oliva, Jeffrey Quispe, Khizar Nasir, Gabriel Prieto

Product Owner: Sam Malloy, Mowafak Allaham, Ayse Lokmanoglu

Instructor: *Masoud Sadjadi*

Problem Definition

Misinformation has a great impact on the public since people are obtaining their information mostly from social media and other online sources. Navigating this chaotic space is overwhelming. People unknowingly spread information from these questionable sources, magnifying the problem. One of the most concerning areas of misinformation is climate change.

In order to shed light on the issue, our team has compiled engagement data related to climate news articles from Newswhip. Using this data we are able to analyze trends and point out topics of concern.

Use Case

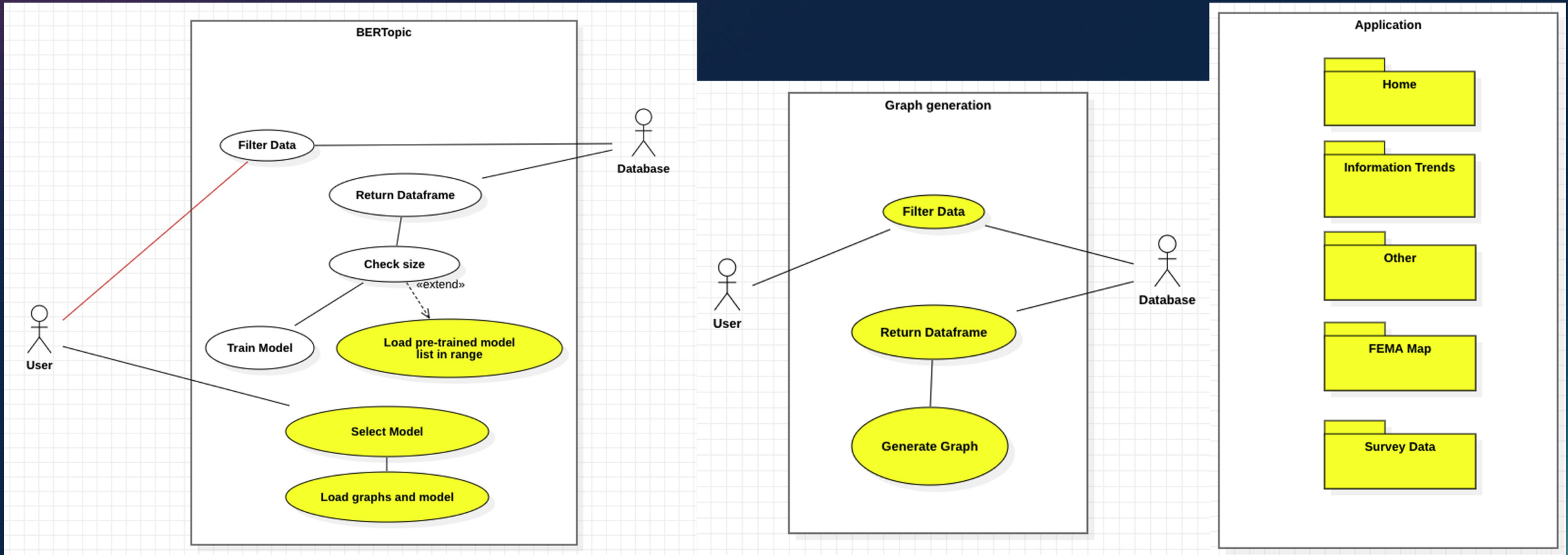
The monitoring and mitigating information project aims to spread awareness about climate misinformation and the public perception of climate related issues.

User's can focus on multiple sources and filter the data according to their needs. This data could be filtered by engagement type, publication date, and top articles. That data can then be analyzed using pie charts, histograms, bubble charts and line graphs.

We used BERTopic to cluster the data into meaningful groups allowing us to find topics related to a search term and thus the articles related to that topic.

Various climate disaster types can also be found on a United States map with the number of incidents separated by states. Estimates of the public's opinion on certain climate questions can also be seen on the survey data page. A breakdown of public interest into climate news can be found on the climate relevancy page.

Project Management



User Stories

User Story 1 - Database Team

User Story 2 - Efficiency Team

User Story 3 - Feature Team

User Story 4 – UI/UX Team

User Story 5 – Machine Learning Team

User Story 6 - Scraper (Gabriel & Roberto)

User Story 7 - Database Queries (Colleene, Brian, and Roberto)

User Story 8 - Anomaly detection (Brian, Roberto, Colleene, Gabriel, Javier) IQR

User Story 9 - Date slider (Done)

User Story 10 – Theme changer (Khizar, Nathan, Jeffrey)

User Story 11 - General UI adjustments to overall theme. (Khizar, Nathan, Jeffrey)

User Story 12 - Survey data UI (In a week)

User Story 13 - Proportion to overall engagement (Roberto, KyleG)

User Story 14 - Informational text box (Javier, KyleZ, Roberto)

User Story 15 - Machine learning model (Brian and Colleene)

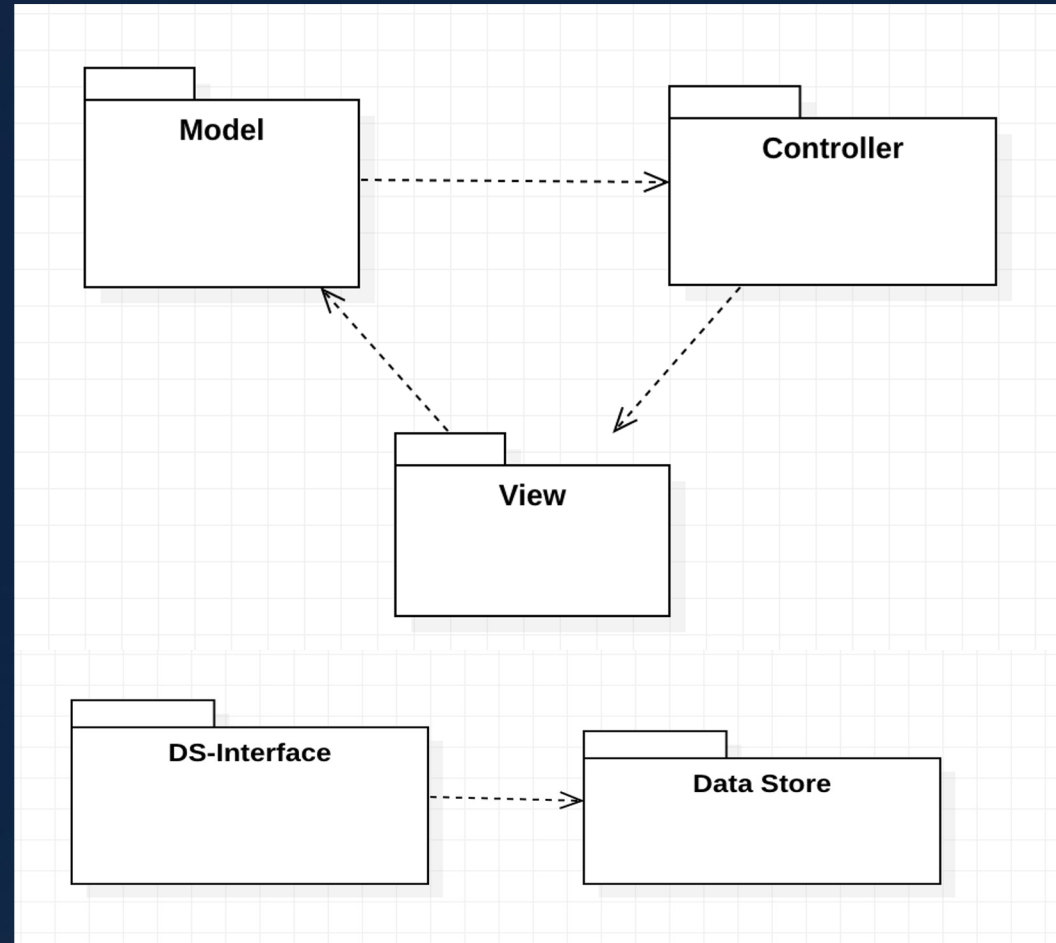
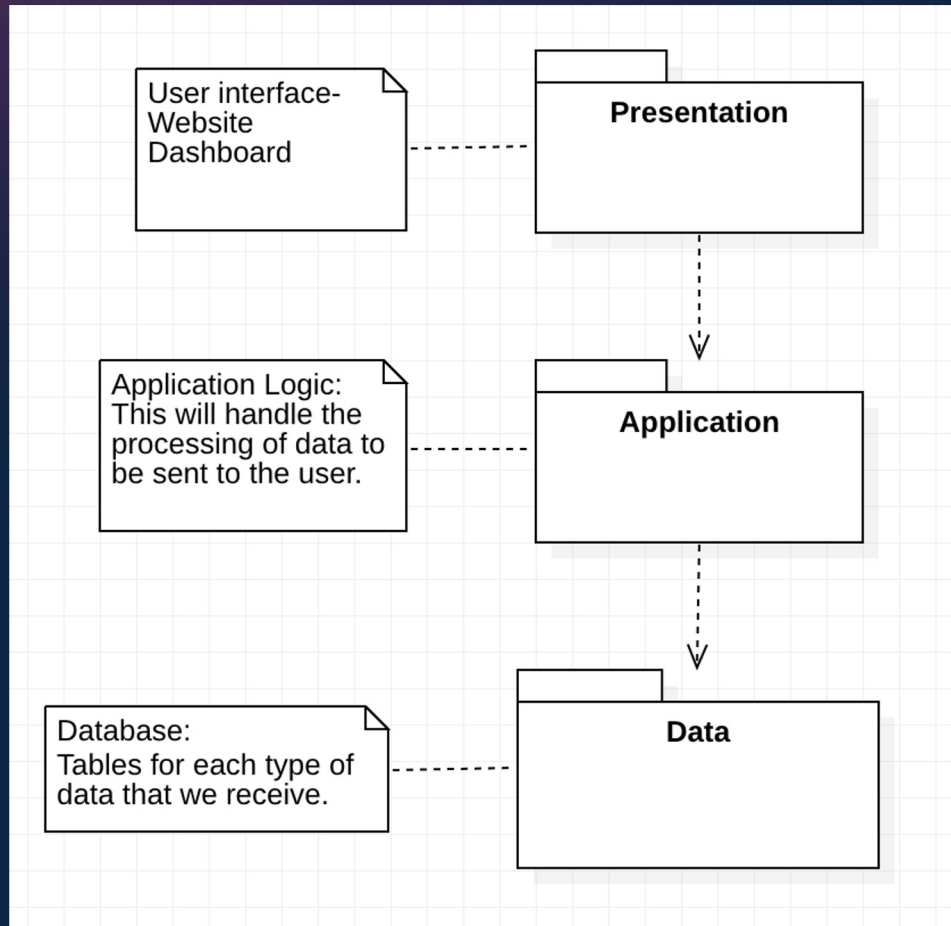
User Story 16 - Sql injection attack (Roberto, Kyle G, Colleene)

User Story 17 - Automate setup environment

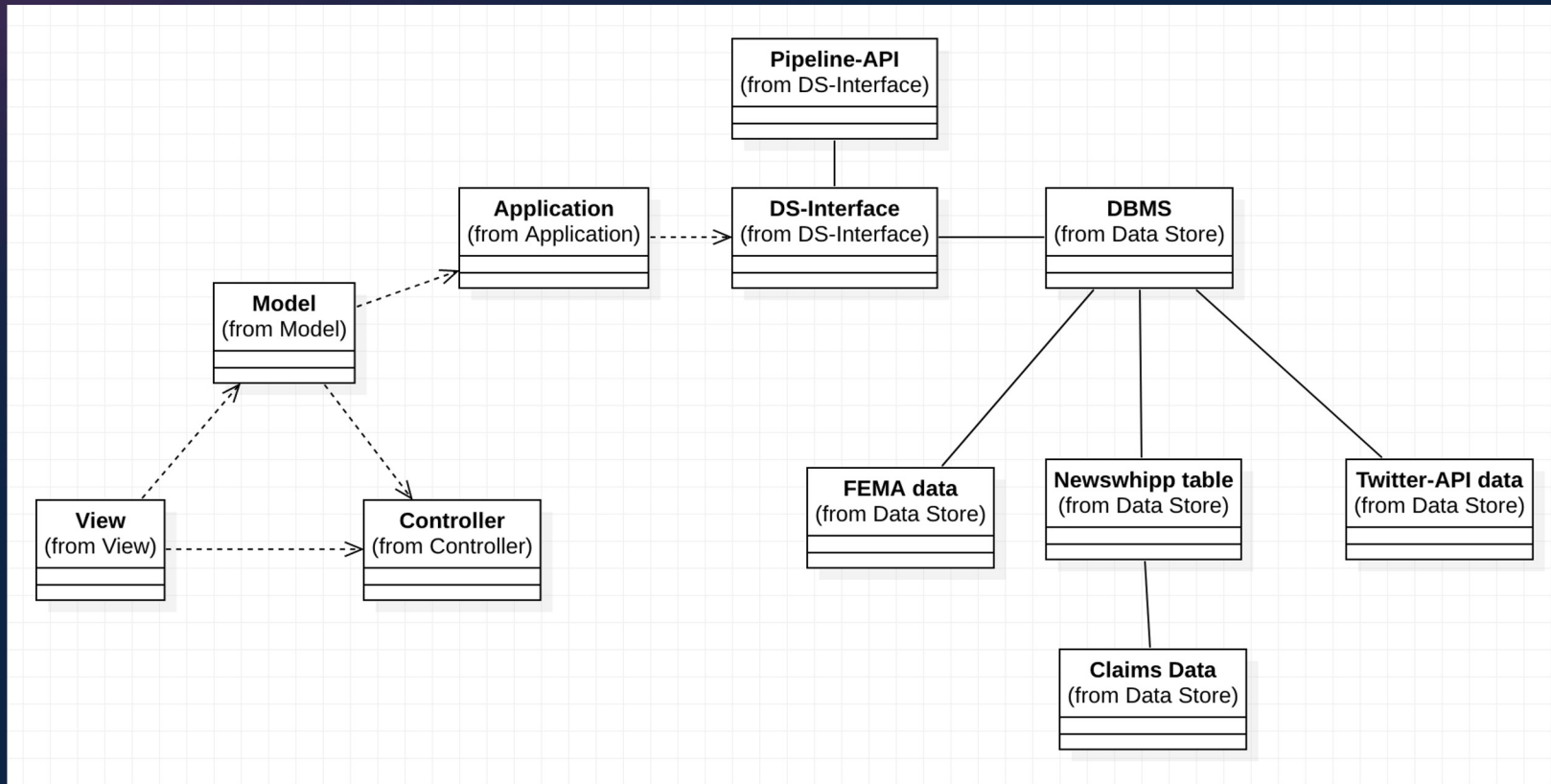
User Story 18 - BertTopic (Brian and Colleene and Roberto)

User Story 19 - MBFC Graph (Colleene, Brian, and Roberto)

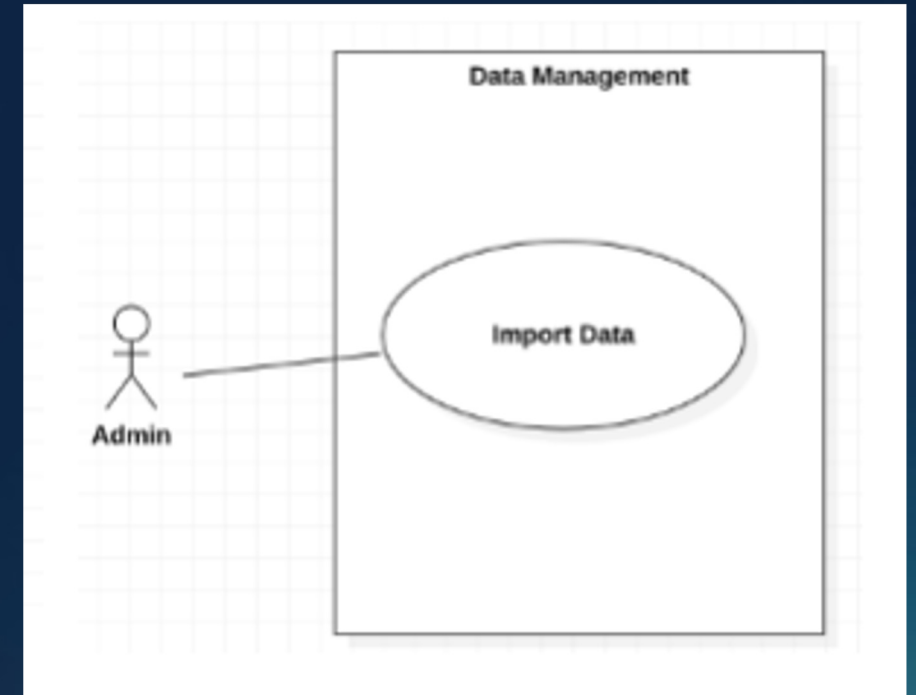
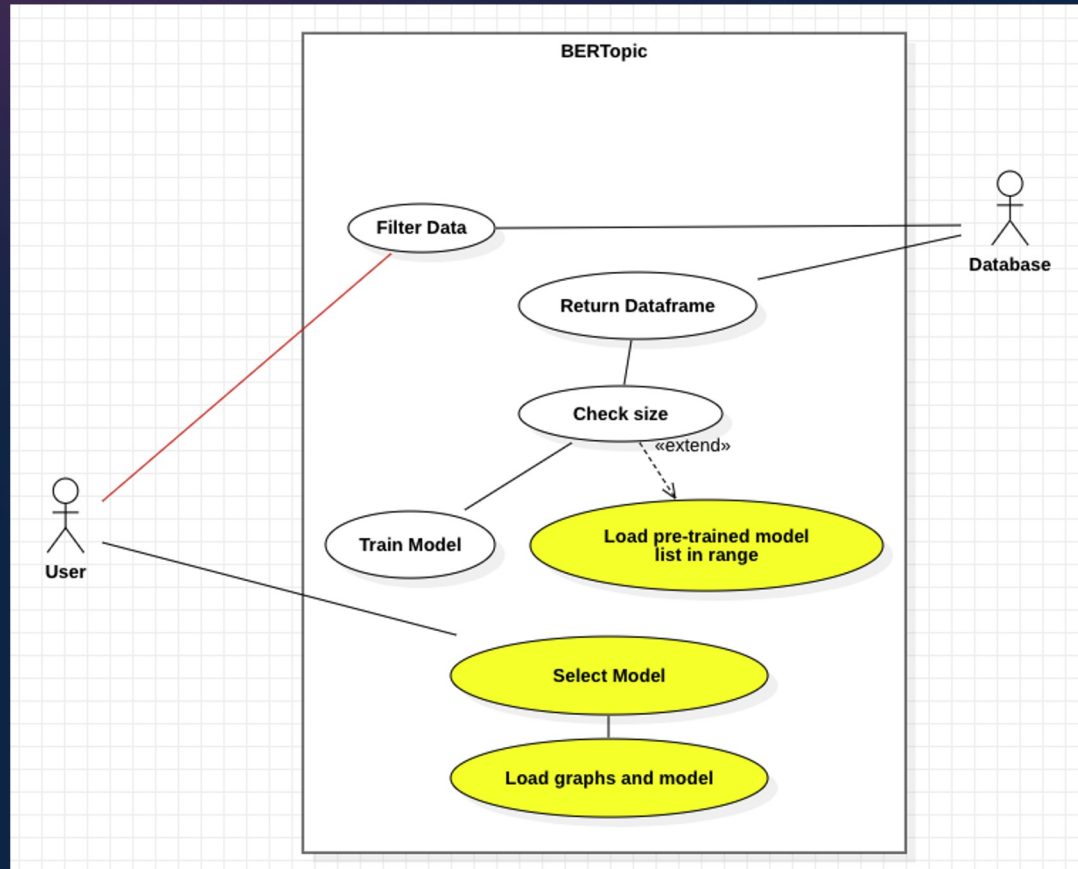
Architectural Patterns



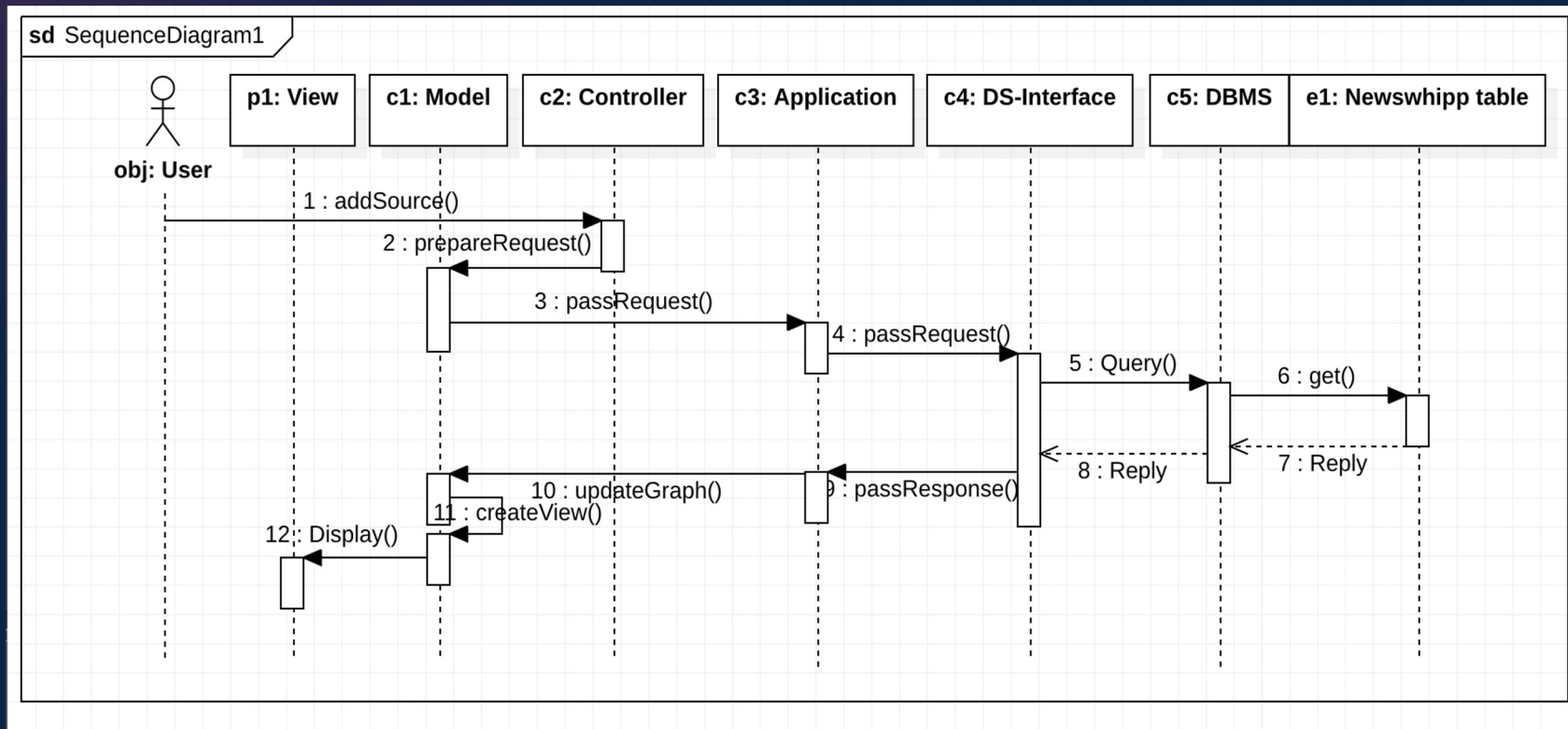
Minimal Class Diagram



Use Case Diagrams EX:



Sequence Diagram



Entity Relationship Diagram



Entities in the database:

1. facebook
2. twitter
3. linkedin
4. pinterest
5. source
6. article

Primary keys are uuid for:

1. facebook
2. twitter
3. linkedin
4. pinterest
5. article

Primary key for source is source_id

All entities have relationship with article

At the moment we are only using facebook and twitter in the app

The database reduces redundancy

Code Breakdown-Database

The script generates a PostgreSQL database from a csv file using psycopg2, SQLAlchemy and pandas

```
#Load env variables
load_dotenv()

Newswhip_Data_Path = os.getenv('Newswhip_Data_Path')
user = os.getenv('user')
password = os.getenv('password')
port = os.getenv('port')
host = os.getenv('host')
database = os.getenv('database').lower()

# set up connection to create database
try:
    conn = psycopg2.connect(user=user, password=password, host=host, port=port)
except:
    print ("I am unable to connect to the database")
cursor = conn.cursor()
conn.set_isolation_level(ISOLATION_LEVEL_AUTOCOMMIT)
cursor.execute(f'DROP DATABASE IF EXISTS {database}')
cursor.execute(f'CREATE DATABASE {database}')
cursor.close()
```

Loads the credentials and the path to the csv file stored in the .env file to make a connection using psycopg2 to a database if already exists, if not then we create a new database

```
# engine set up
# dialect+driver://username:password@host:port/database
alchemyEngine = create_engine(f"postgresql+psycopg2://{user}:{urllib.parse.quote_plus(password)}@{host}:{port}/{database}",
                              pool_recycle=3600,
                              echo=True)
```

Creates an SQLAlchemy engine using the loaded values

```
def create_source_pkey_article_fkey(df, entity, key_atts):
    if entity not in ['source', 'article']:
        return
    df['source_id'] = df[key_atts].agg('/'.join, axis=1)
    if entity == 'article':
        df.drop(columns=key_atts, inplace=True)
    if entity == 'source':
        df.drop_duplicates(subset=['source_id'], inplace=True)
```

```
create_source_pkey_article_fkey(df_temp,
                                entity,
                                ['source.publisher', 'source.country_code', 'source.language'])
```

Creates a primary key for the source entity and a foreign key for the article entity

```
# convert dataframe to database
def df2sql(df, postgresqlTable, alchemyEngine):
    df.columns = [c.replace('.', '_') for c in df.columns]

    try:
        df.to_sql(postgresqlTable, alchemyEngine, if_exists='replace')
    except Exception as ex:
        print(ex)
    else:
        print("PostgreSQL Table %s has been created successfully."%postgresqlTable)
```

`df2sql(df_temp, entity, alchemyEngine)`

Creates the database using the to_sql function in pandas passing the created SQLAlchemy engine

```
# add primary and foreign key constraints
with alchemyEngine.connect() as con:
    con.execute(f'ALTER TABLE {entity} DROP CONSTRAINT IF EXISTS {entity}_pkey;')
    if entity == 'source':
        con.execute(f'ALTER TABLE {entity} ADD PRIMARY KEY (source_id)')
    else:
        con.execute(f'ALTER TABLE {entity} ADD PRIMARY KEY (uuid)')

    if entity == 'article':
        con.execute(f'ALTER TABLE article ADD FOREIGN KEY (source_id) REFERENCES source(source_id)')
        for n, e in zip(nicknames, sm_entities):
            con.execute(f'ALTER TABLE article ADD FOREIGN KEY ({n}_uuid) REFERENCES {e}(uuid)')
```

Add primary and foreign key constraints to the entities.

For the article and all the social media platforms the primary key is the uuid, while source_id is the primary key of source.

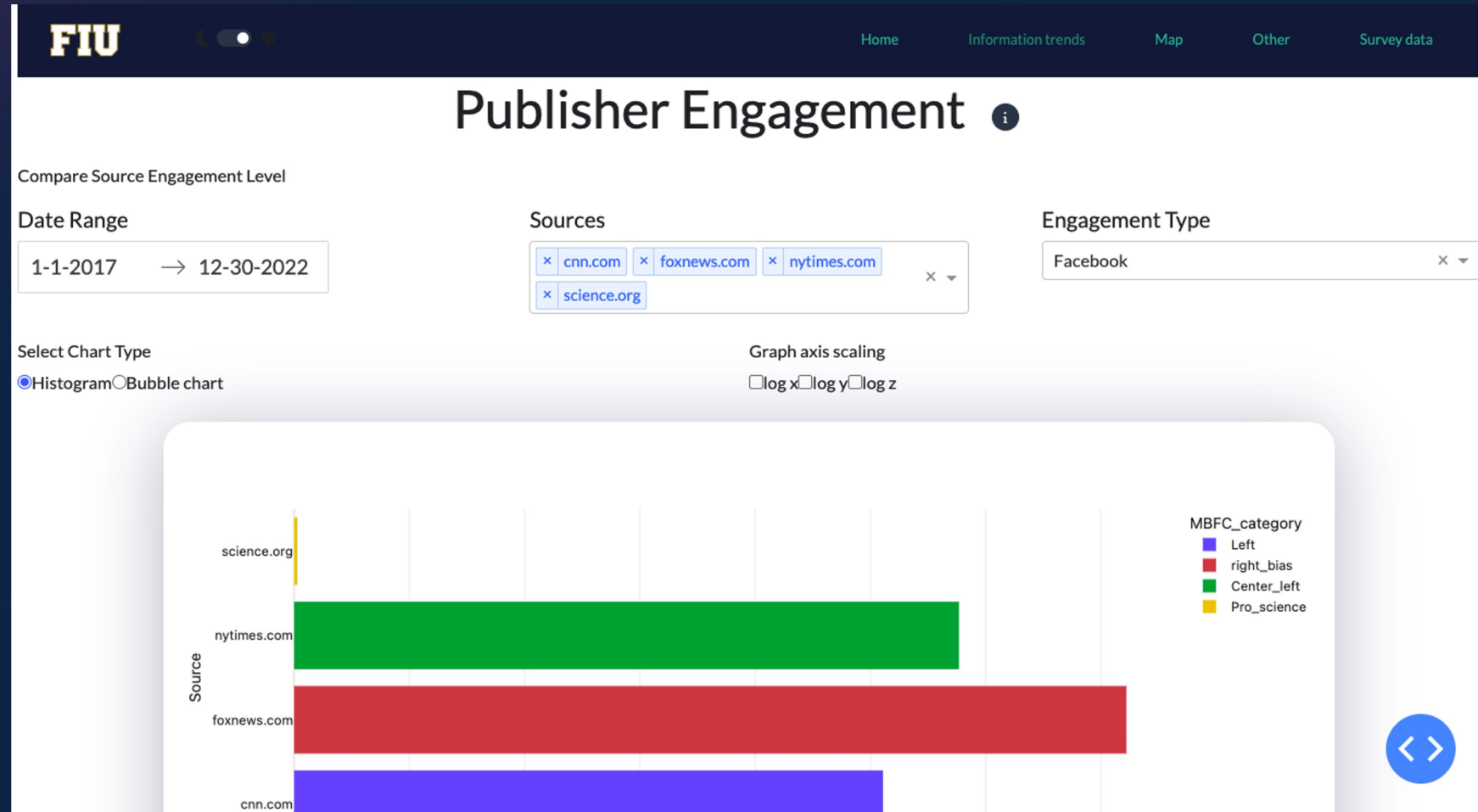
The article entity contains as foreign keys the ids of the social media platforms and the source entity

Code Breakdown -Histogram/Bubble Chart

Application that filters dataframe by time, groups by source and engagement.

This function is called every time the graph is interacted with to regenerate the graph.

The graph will take into account the time selected the, filter by that time, and then further filtered by the sources selected. The final dataframe is then compared to a reference table of MBFC labels to generate the colors.

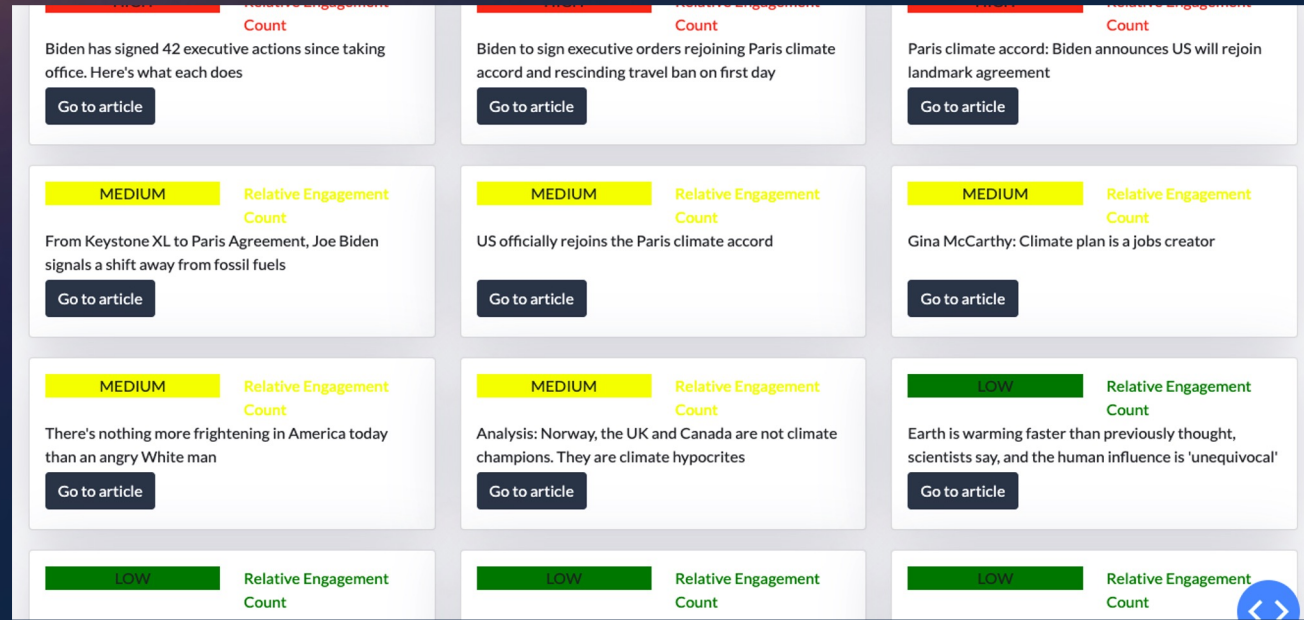


Code Breakdown -Cards



```
def card_main(title, excerpt, link, index, listLength ):
    color='green'
    tag='LOW'
    if index/listLength< 0.10:
        color='red'
        tag='HIGH'
    elif index/listLength< 0.30:
        color='yellow'
        tag='MEDIUM'

    return dbc.Card(
    [
        dbc.CardBody(
            [
                dbc.Row([
                    dbc.Col([
                        html.Div(
                            tag,
                            style={
                                'backgroundColor':color,
                                'textAlign':'center'
                            })
                    ],
                    dbc.Col([
                        html.Div(
                            'Relative Engagement Count',
                            style={
                                'color':color
                            })
                    ])
                ]),
                html.P(
                    title,
                    className="card-text",
                    style={
                        'padding-bottom':'30px',
                    },
                ),
                dbc.Button("Go to article",href=link, color="primary",style={
                    'position':'absolute',
                    'bottom':'20px',
                    'margin-top':'20px'
                }),
            ]
        ),
    ],
    class_name='cardBody'
)
```



Cards are created as a way to visually represent the articles composing a BERTopic cluster

Cards are color coded and sorted according to the engagement level they receive in a facebook or twitter. Red mean high engagement, yellow medium and green means low engagement.

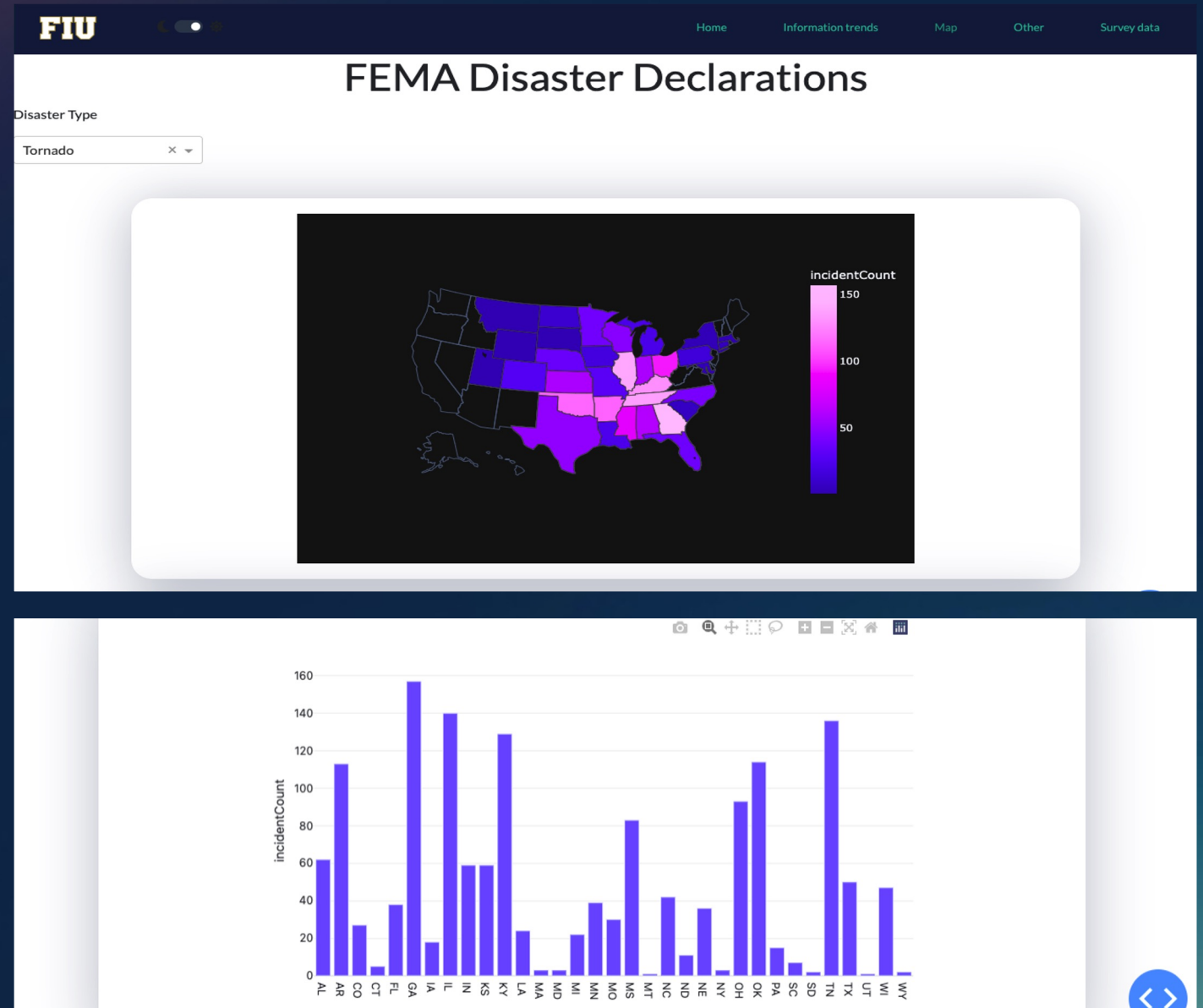
As we can see dash has a component called Card that contains a CardBody object where we can put the tag, engagement, and the headline of the article

We can also go to a specific article by clicking the Go to Article button. At the moment when the button is clicked we navigate to the website right away, but we should add a warning for the user since they are leaving the webapp.

Code Breakdown -FEMA Data

This map represents the disaster declarations that have happened in each state. When hovering over a state. The incident count is shown.

An additional graph shows a comparison of the states by hurricane incident count.



Code Breakdown -FEMA Data



```
app.layout = html.Div([
    html.H1("FEMA Disaster Declarations", style={'text-align' : 'center'}),
    html.P("Disaster Type", style= {'margin-top': '5px', 'color' : 'black'}),
    dcc.Dropdown(id="select_type",
        options=[{"label": "Tornado", "value": "Tornado"},
                  {"label": "Hurricane", "value": "Hurricane"},
                  {"label": "Flood", "value": "Flood"},
                  {"label": "Fire", "value": "Fire"},
                  {"label": "Severe Storms", "value": "Severe Storm(s)"},
                  {"label": "Drought", "value": "Drought"},
                  {"label": "Coastal Storm", "value": "Coastal Storm"},
                  {"label": "Snow", "value": "Snow"},
                  {"label": "Freezing", "value": "Freezing"},
                  {"label": "Severe Ice Storm", "value": "Severe Ice Storm"},
        ],
        multi=False,
        value="Tornado",
        style={'width':"40%"}),

    html.Div(id='output_container', children=[]),
    html.Br(),

    dcc.Graph(id='disaster_map', figure={})
])
```

```
def update_graph(option_selected):

    container = "Disaster Type: {}".format(option_selected)

    dff = disasters_data.copy()
    dff = dff[dff['incidentType'] == option_selected]

    fig = px.choropleth(
        data_frame=dff,
        locations='state',
        locationmode='USA-states',
        scope="usa",
        color='state',
        hover_data=['state', 'incidentCount'],
        color_continuous_scale=px.colors.sequential.YlGnBu,
        labels={'incidentType': 'Incident Type'},
        template='plotly_dark'
    )

    return container, fig
```

The code used to create the map.

Code on the left refers to the html creation, the right side refers to the graph creation

Code Breakdown -Scraper

This function iterates over each url to be scraped and adds the collected data to a dataframe.

```
UNIX_HOUR=300 #3600 is an hour, this is 5 minutes
print('total links',total)

def scrape(list_of_urls,startP,endP,fileName):
    if endP == 'last':
        list_of_urls=list_of_urls[startP:]
    else:
        list_of_urls=list_of_urls[startP:endP]

    hourCount=1
    count=0
    rows = []
    # print(len(list_of_urls))
    for link in list_of_urls:
        count+=1
        try:
            a = Article(url="%s" % (link), language='en')
            a.download()
            a.parse()

            author = a.authors
            text = a.text
            title = a.title

            if (time.time()-start)>hourCount*UNIX_HOUR:
                print(f'fileName: {count} of {len(list_of_urls)} completed')
                hourCount+=1

            # print(f'\n\nNewspaper***\ntitle:{title}\n\ntext:{text}')
            text=text.replace('\n','')

            row = {'url':link,
                  'author':author,
                  'textNW':text,
                  'title': title}
```

The output of each thread is sent to the thread out folder and once the threads are joined a final data frame is created.

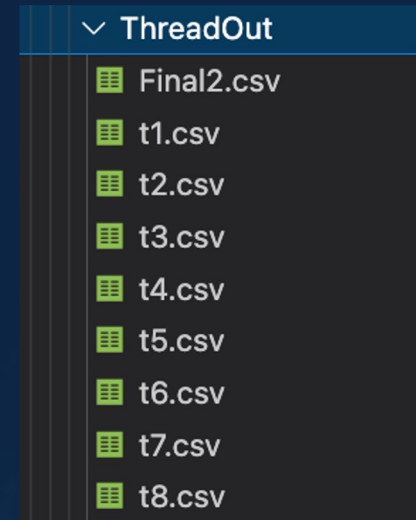
Each thread is passed the scrape function to be executed. The argos passed to the function is the subsection which the threads will process.

```
df_v1 = pd.DataFrame(rows)

dfmaster = dfmain.merge(df_v1, left_on='link', right_on='url')

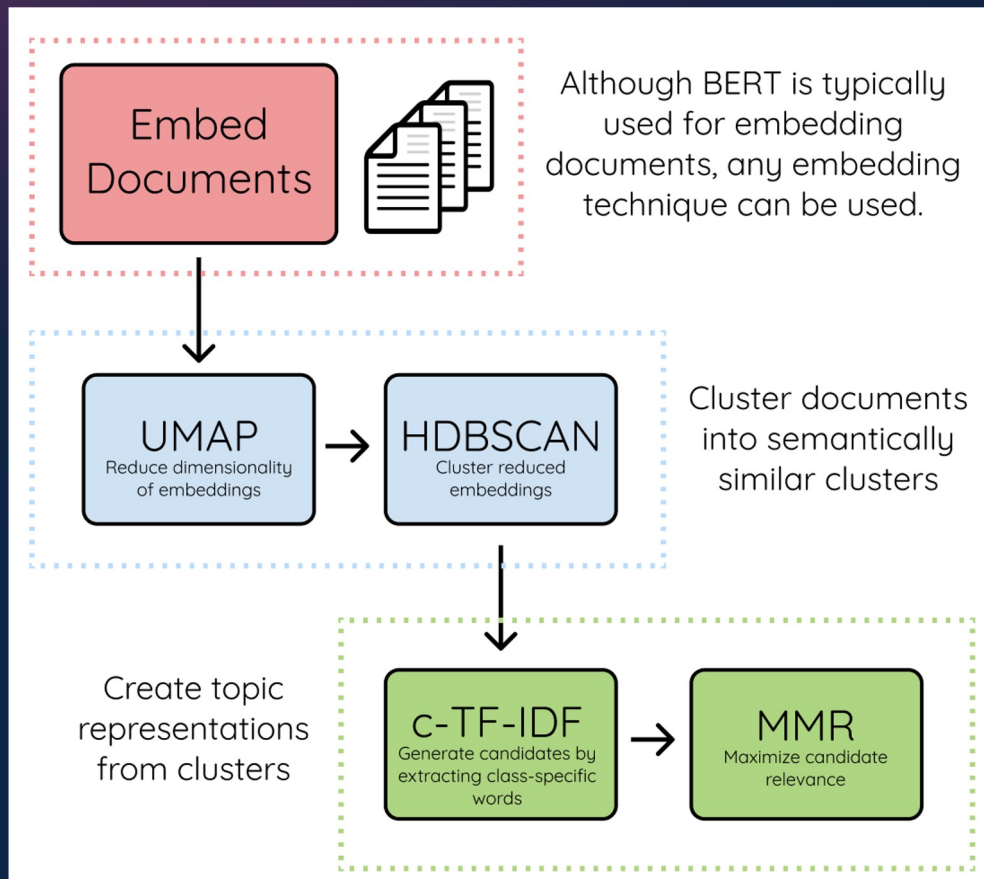
dfmaster.to_csv(PATH+fileName +'.csv')

t1 = threading.Thread(target=scrape,name='t1', args=(list_of_urls,0,total//8,'t1'))
t2 = threading.Thread(target=scrape,name='t2', args=(list_of_urls,total//8,total//4,'t2'))
t3 = threading.Thread(target=scrape,name='t3', args=(list_of_urls,total//4,3*total//8,'t3'))
t4 = threading.Thread(target=scrape,name='t4', args=(list_of_urls,3*total//8,total//2,'t4'))
t5 = threading.Thread(target=scrape,name='t5', args=(list_of_urls,total//2,5*total//8,'t5'))
t6 = threading.Thread(target=scrape,name='t6', args=(list_of_urls,5*total//8,3*total//4,'t6'))
t7 = threading.Thread(target=scrape,name='t7', args=(list_of_urls,3*total//4,7*total//8,'t7'))
t8 = threading.Thread(target=scrape,name='t8', args=(list_of_urls,7*total//8,'last','t8'))
```



Code Breakdown -BERTopic Model

BERTopic is a topic modeling technique that leverages transformers and c-TF-IDF to create dense clusters allowing for easily interpretable topics whilst keeping important words in the topic descriptions



The algorithm contains 3 stages:

1. Embed documents

- Extract document embeddings with BERT or any other embedding technique
- "all-MiniLM-L6-v2" as default

2. Cluster Documents

- UMAP to reduce the dimensionality of embeddings
- HDBSCAN to cluster reduced embeddings and create clusters of semantically similar documents

3. Create topic representation

- Extract and reduce topics with c-TF-IDF
- Improve coherence and diversity of words with Maximal Marginal Relevance

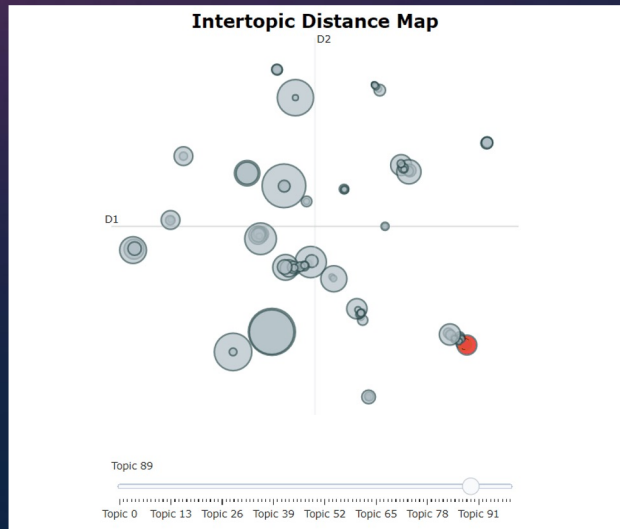
Code Breakdown

To use BERTopic we need to install it and then import it like this

```
! pip install bertopic
from bertopic import BERTopic
```

Instantiate a BERTopic object and fit/transform a list of documents

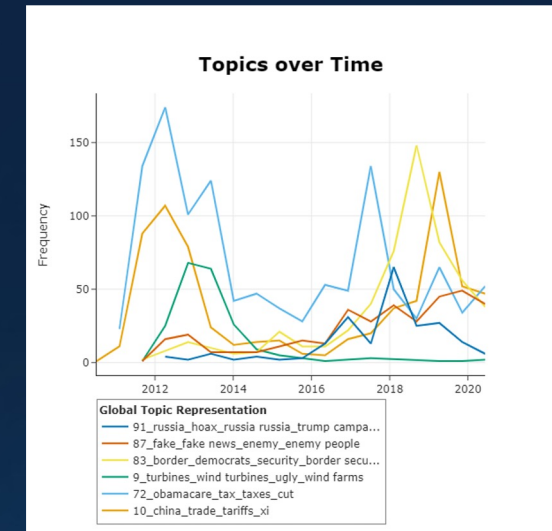
```
topic_model = BERTopic()
topics, probs = topic_model.fit_transform(docs)
```



- Embed the c-TF-IDF representation of the topics in 2D using Umap
- Visualize the two dimensions using plotly
- **.visualize_topics** to create a 2D representation of the topics.



- Create bar charts out of the c-TF-IDF scores for each topic representation.
- Compare topic representations to each other.
- To visualize this hierarchy, run the function **.visualize_barchart()**



- Show the frequency of topics over time
- Hover over the points to show the time-specific topic representations.
- Call **.visualize_topics_over_time** with a topics over time object created when using the `model.topics_over_time` function

Code Breakdown -Aggregated Data



Graph represents the proportion of Climate news engagement to total news engagement.
The graph can be switched to show the mean engagement of climate articles as well.

The code below groups the data by month and creates the sum of engagement and total article count for each month.

```
def CreateAggregates(filename):  
    #Read Csv and index by date_time  
    df = pd.read_csv(filename,encoding='Latin1')  
    df = df[NEEDED_COLUMNS].copy()  
    df.index = pd.to_datetime(df['date_time'])  
  
    #group data by month and sum up articles that occurred each month using .count()  
    countDF= df.groupby(pd.Grouper(freq='M')).count()  
    countDF= countDF[['date_time']].copy()  
    countDF.rename(columns={'date_time':'articlecount'},inplace=True)  
  
    #group data by month and sum up engagement on a separate DataFrame  
    df= df.groupby(pd.Grouper(freq='M')).sum()  
    #Merge the two data frames  
    df=df.join(countDF)  
  
    return df
```

Code Breakdown -UI/UX



Dash is a python framework created by plotly for creating interactive web applications.

Dash is written on the top of Flask, Plotly.js and React.js.

With Dash, you don't have to learn HTML, CSS and Javascript in order to create interactive dashboards, you only need python.

Dash applications are made up of 2 building blocks :

1. Layout
2. Callbacks

```
app = dash.Dash(__name__)

app.layout = html.Div([
    html.H1("NewsWhipPublishers", style={'text-align': 'center'}),
    html.P("Publisher", style={'margin-top': '5px', 'color': 'black'}),
    dcc.Dropdown(id="select_publisher",
                 options= dropDownPublishers,
                 searchable=True,
                 multi=False,
                 value="cnn.com",
                 style={'width': "40%"}),

    dcc.Dropdown(id="select_topic",
                 options= dropDownTopics,
                 searchable=True,
                 multi=False,
                 value="ALL",
                 style={'width': "40%"}),

    dcc.Dropdown(id="select_percentage",
                 options= dropDownPercentages,
                 searchable=True,
                 multi=False,
                 value="10",
                 style={'width': "40%"}),

    html.Div(id='output_container', children=[]),
    html.Br(),

    dcc.Graph(id='publisher_histog', figure={}),

])
```

Layout describes the look and feel of the app, it defines the elements such as graphs, dropdowns, buttons, cards, sliders, ect.

The placement, size, and color of these elements can also be specified here.

Dash contains **Dash HTML components** such as headings, paragraph, images, ect.

```
@app.callback([
    Output(component_id='output_container', component_property='children'),
    Output(component_id='publisher_histog', component_property='figure')],
    [Input(component_id='select_publisher', component_property='value'),
     Input(component_id='select_topic', component_property='value'),
     Input(component_id='select_percentage', component_property='value')])

def update_graph(option_selected, topic_selected, percentage_selected):

    container = "Publisher Source: {}".format(option_selected)
    print(option_selected)
    print("topic_selected", topic_selected)
    print("percentage_selected", percentage_selected)
    dff = prelimEngageDF

    ###Splice by source
    dff = dff[dff['source.publisher'] == option_selected]

    ###Splice by keyword
    if(topic_selected != 'ALL'):
        dff = dff[dff['keywords'].astype(str).str.contains(topic_selected)]

    print(dff['keywords'])
    dff = dff.sort_values(by='fb_data.total_engagement_count', ascending=False)
    dff = dff.head(int((len(dff)*(int(percentage_selected)/100))))

    fig = px.histogram(
        dff,
        y='headline',
        x='fb_data.total_engagement_count',
    )

    fig.update_xaxes(title_text='Engagment Count')
    fig.update_yaxes(title_text='Article')

    fig.update_traces(texttemplate="%{}")

    return container, fig
```

Callbacks are used to bring interactivity to the dash applications.

These are the functions we use to define the activity that would happen on clicking a button or a dropdown.

A callback is initialised using `@app.callback()`, which is followed by a function definition.

Inside the callback decorator we specified the outputs and inputs of the function

Dashboard



What is Monitoring and Mitigating Misinformation

The monitoring and mitigating information project aims to spread awareness about climate misinformation and the public perception of climate related issues. User's are able to interact with the engagement types of different sources and compare them visually with either a histogram or a bubble chart. There is also a chart comparing the engagement level of different articles from a publisher and links to the categorized articles below. Various climate disaster types can also be found on a United States map with the number of incidents separated by states. Estimates of the public's opinion on certain climate questions can also be seen on the survey data page

Our Data Sources

The survey data of climate opinions was conducted by the Yale Program on Climate Change Communication and their most recent climate opinion map can be found at <https://climatecommunication.yale.edu/visualizations-data/vcom-us/Disaster> data was collected by political affiliation based on what political values the article is seen to follow. Monitoring and the Federal Emergency Management Agency which is part of the Department of Homeland Security Mitigating Misinformation follows engagement in regards to these articles, ordering the articles and helps people with disaster problems. The article and publisher engagement data was found through NewsWhip, a real-time media monitoring platform.

Our Process

Monitoring and Mitigating Misinformation scrapes the most frequently visited news sources for articles relating to climate change. In this regard, an algorithm is applied to categorize articles into political affiliation based on what political values the article is seen to follow. Monitoring and the Federal Emergency Management Agency which is part of the Department of Homeland Security Mitigating Misinformation follows engagement in regards to these articles, ordering the articles from most to least engaged with articles.

Our Data Sources

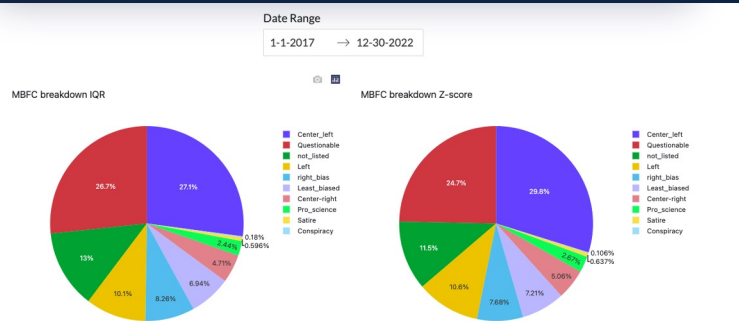
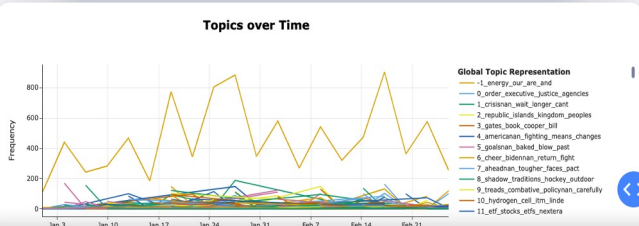
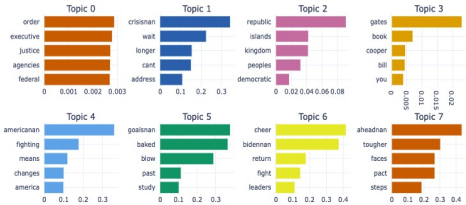
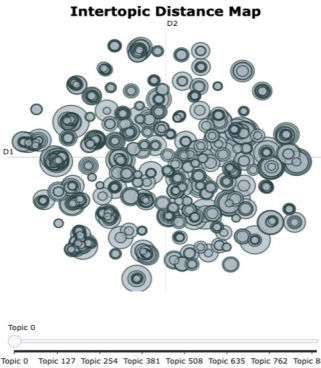
sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad

Have any questions?

Can't find the answers you're looking for?

Get in touch

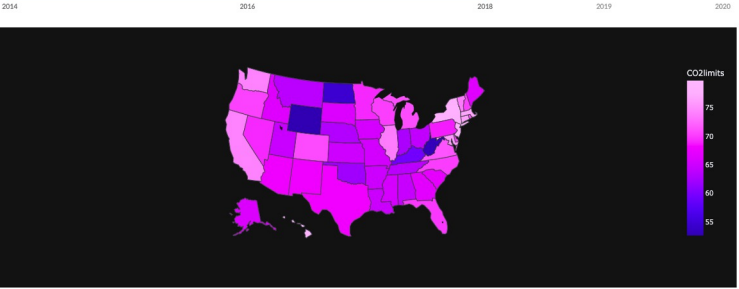
BERTopic Model



Climate Survey Data

Questions asked

Estimated percentage who somewhat/strongly support setting strict limits on existing coal-fire power plants



Summary

We compiled the work from the previous capstone iteration and turned it into a DASH multi page application with analytical functions and visual aids in the form of dynamic graphs. We used BERTopic to cluster the articles in meaningful groups to find relevant documents to the user's searched term. All the Newswhip data static files that were being used as a reference for the scripts were put into a Postgres database. We tested the database by building a proof of concept using optimized queries.

We created a multithreaded processing scraper using beautiful soup. All UI features were added during this terms iteration. BERTopic used to cluster articles based on their text and headlines. Github pages were deployed and used for collaboration. Database was created, connected to and queried to from application. Further iterations will make an application that is more depended on the database.

Next Steps...

- The next steps for the project are to deploy a postgres server at FIU. There is a lot of data and it needs to be properly housed.
- The BERT models need to be tweaked as they are subject to over classification. (Stop words need to be cut from the training data)
- Claims algorithm needs to be implemented. The script is being analyzed by the PO and will begin being used to extract the claims made by articles.
- Queries need to replace dataframe manipulations for static data. Some have been done, but many more need to be completed.
- Pipeline needs to be created, so that new articles can be inputted every couple of weeks and real time anomaly detection can be implemented.
- Anomaly Detection that detects abnormalities in news data based on historical trends.
Example: Uptrend in engagement with questionable sources. Or an uptrend in articles that make a specific claim from a specific political bias.
- Correlation of News article data to Survey Data and Fema disaster Data.
- Scrape the rest of the data for the years 2017-2020. Run BERTopic models on those years.