Overview

This project was created with the goal of analyzing feelings about the weather in a chosen city over the last 7 days and comparing it to weather observations over the same period. The tools used to implement this project include the Twitter API (with the help of tweepy), National Weather Service API (with the help of noaa_sdk), BM25 ranking for eliminating less relevant results returned from Twitter, VADER sentiment libraries to gauge the sentiment in the tweets, Flask to help bridge the gap with the front end, and D3.js for front end data handling and visualizations.

Code Walkthrough

app.py

The beginning block of code handles importing libraries, defining the Flask app, declaring a class for data stores (described later) that will be sent to our front end, and setting up a route for post requests.

```
# importing all necessary modules
import tweepy
from noaa_sdk import NOAA
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
import pandas as pd
import numpy as np
from rank_bm25 import BM25Okapi
import re
from flask import Flask, render_template, request, jsonify

# Declare application
app= Flask(__name__)

# Declare data stores
class DataStore():
    CityName=None
    WeatherVals=None
    WeatherObs=None
    #CloudLyr=None
    SentVals=None
    TweetTerms=None
data=DataStore()

# Define main code
@app.route("/main",methods=["GET","POST"])

@app.route("/",methods=["GET","POST"])
```

The homepage function handles everything on the analysis that need to be done for our home (and only) page on the front end.

```
def homepage():
```

We start by reading in the city name submitted from the user, and using our geocode dictionary to grab the zip code of the city.

The cities included in our selection list for this project are the central cities for the 25 most populated primary statistical areas (PSA) in the United States.

The specific zip codes for those cities come from the address of the largest airport in each PSA. This ensures the NWS API pulls observations from reliable weather stations, which are typically located at international airports. New York, Washington DC, and Philadelphia were eliminated as there were issues with the generated results from those cities, and there wasn't time to resolve them.

```
    # Get city selected from user submission, default is Chicago
    CityName = request.form.get('City_field','Chicago')

    q = CityName.lower()

    geocode = {
        #"new york" : '11430',
        "los angeles" : '90045',
        "chicago" : '60666',
        #"washington dc" : '22202',
        "san francisco" : '94128',
        "boston" : '02128',
        "dallas" : '75261',
        "houston" : '77032',
        #"philadelphia" : '19113',
        "atlanta" : '30320',
        "miami" : '33142',
        "detroit" : '48174',
        "phoenix" : '85034',
        "seattle" : '98158',
        "orlando" : '32827',
        "minneapolis" : '55450',
        "denver" : '80249',
        "cleveland" : '44135',
        "san diego" : '92101',
        "portland" : '97202',
        "tampa" : '33614',
        "st louis" : '63145',
        "charlotte" : '28208',
        "salt lake" : '84122',
        "sacramento" : '95837',
    }

    zip = geocode[q]
```

Now we define our object for grabbing weather data utlizing the noaa_sdk, developed by Paulo Kuong (https://github.com/paulokuong/noaa), which serves as a generic wrapper for the NWS API.

We then pull observation data from every hour over the last 7 days (the default for the SDK) and place the observations we care about in the defined data frame, converting every value to the desired unit of measurement.

```
    n = NOAA()

    # each row will record an hourly observation
    obs = pd.DataFrame(columns=['timestamp', 'textDescription', 'temperature',
'windSpeed', 'visibility',
                        'precipitationLastHour', 'relativeHumidity', 'windChill',
'heatIndex'],#, 'cloudLayers'],
                        index=range(1, len(list(n.get_observations(zip,'US')))))

    # get NWS observations from last 7 days for city using zip code
    observations = n.get_observations(zip,'US')
    rowNum = 0

    for observation in observations:
        obs['timestamp'][rowNum] = observation['timestamp']
        obs['textDescription'][rowNum] = observation['textDescription']
        #print(observation['textDescription'])
        if type(observation['temperature']['value']) == int or
type(observation['temperature']['value']) == float:
            # convert to Fahrenheit
            obs['temperature'][rowNum] = float(observation['temperature']
['value'])*9/5+32
        if type(observation['windSpeed']['value']) == int or
type(observation['windSpeed']['value']) == float:
            # convert to mph
            obs['windSpeed'][rowNum] = float(observation['windSpeed']['value']) *
0.621371
        if type(observation['visibility']['value']) == int or
type(observation['visibility']['value']) == float:
            # convert to miles
            obs['visibility'][rowNum] = float(observation['visibility']['value'])
* 0.000621371
        if type(observation['relativeHumidity']['value']) == int or
type(observation['relativeHumidity']['value']) == float:
            # percent humidity
            obs['relativeHumidity'][rowNum] = observation['relativeHumidity']
['value']
        if type(observation['windChill']['value']) == int or
type(observation['windChill']['value']) == float:
            # convert to Fahrenheit
            obs['windChill'][rowNum] = float(observation['windChill']
['value'])*9/5+32
        if type(observation['heatIndex']['value']) == int or
type(observation['heatIndex']['value']) == float:
            # convert to Fahrenheit
```

```
            obs['heatIndex'][rowNum] = float(observation['heatIndex']
['value'])*9/5+32
        #if len(observation['cloudLayers']) > 1:
            # amount of cloud cover
        #    obs['cloudLayers'][rowNum] = observation['cloudLayers'][0]['amount']
        rowNum = rowNum + 1
```

We then take those values and create our data stores that will be sent to the front end. This includes one with the min, max, and average for each numerical weather observation (WeatherVals) and another with the text descriptions for the weather, along with the count of each of these terms (WeatherObs)

```
    # pull data from observations to create data we want to display in UI
    try:
        descriptions = obs['textDescription'].tolist()
        list_of_weather_obs = []
        for row in descriptions:
            items = re.split(" and ", row)
            for item in items:
                if item != '':
                    list_of_weather_obs.append(item)
        weather_obs = pd.Series(list_of_weather_obs).value_counts()
    except:
        weather_obs = pd.Series(dtype='int')
    try:
        temp = "Temperature (F):  Max: {} Avg: {} Min: {}".format(
                int(np.nanmax(obs['temperature'])),
                int(np.nanmean(obs['temperature'])),
                int(np.nanmin(obs['temperature'])))
    except:
        temp = "Temperature NA"
    try:
        wind_sp = "Wind Speed (mph):  Max: {} Avg: {} Min: {}".format(
                int(np.nanmax(obs['windSpeed'])),
                int(np.nanmean(obs['windSpeed'])),
                int(np.nanmin(obs['windSpeed'])))
    except:
        wind_sp = "Wind Speed NA"
    try:
        vis = "Visibility (miles):  Max: {} Avg: {} Min: {}".format(
                int(np.nanmax(obs['visibility'])),
                int(np.nanmean(obs['visibility'])),
                int(np.nanmin(obs['visibility'])))
    except:
        vis = "Visibility NA"

    try:
        hum = "Rel Humidity (percent):  Max: {} Avg: {} Min: {}".format(
                int(np.nanmax(obs['relativeHumidity'])),
                int(np.nanmean(obs['relativeHumidity'])),
                int(np.nanmin(obs['relativeHumidity'])))
    except:
```

```
        hum = "Rel Humidity NA"
    try:
        wind_ch = "Wind Chill (F):  Max: {} Avg: {} Min: {}".format(
                int(np.nanmax(obs['windChill'])),
                int(np.nanmean(obs['windChill'])),
                int(np.nanmin(obs['windChill'])))
    except:
        wind_ch = "Wind Chill NA"
    try:
        heat_ind = "Heat Index (F):  Max: {} Avg: {} Min: {}".format(
                int(np.nanmax(obs['heatIndex'])),
                int(np.nanmean(obs['heatIndex'])),
                int(np.nanmin(obs['heatIndex'])))
    except:
        heat_ind = "Heat Index NA"

    #try:
    #    cloud_lyr = obs['cloudLayers'].value_counts()
    #except:
    #    cloud_lyr = pd.Series({'NA': 'NA'})

    data.WeatherObs = []

    # get value counts to be used for bar chart
    for name, weight in weather_obs.items():
        data.WeatherObs.append({'name': name, 'weight' : weight})

    data.WeatherVals = dict({'names' : ['temp', 'wind_sp', 'vis', 'hum',
'wind_ch', 'heat_ind'],
                             'values': [temp, wind_sp, vis, hum, wind_ch,
heat_ind]})
    #data.CloudLyr = dict({'names': list(cloud_lyr.axes[0]), 'weights' :
cloud_lyr.values.tolist()})
```

With our weather observations set to send, we now prepare to pull data from the Twitter API with the help of tweepy. We need a bearer token to access the API. This can be generated using a Twitter developer account.

We will create a corpus containing all the Twitter documents, given a query, and a parallel id array to keep track of users. We also create the scores data frame that will contain the results of a BM25 scoring function.

```
    # get tweepy bearer token from txt file
    with open('bearer_token.txt') as bt:
        betk = bt.readline()

    # create client with bearer token
    client = tweepy.Client(bearer_token=betk)

    # Get tweets that match queries
    # -is:retweet means I don't wantretweets
    # lang:en is asking for the tweets to be in english
```

```
    # for entire corpus from queries
    corpus = []
    # for tweet ids
    ids = []
    # for bm25 ranking scores
    scores = pd.DataFrame(columns=['id', 'document', 'score'])
```

The query generated for Twitter documents is very simple, 7 two word combinations, first word is the city, second word is a different weather term for each combo. This isn't a perfect system, as we get interesting results for some cities (i.e. philadelphia sunny returns mostly results about a tv show and buffalo hot is mostly about the sauce). For each of these tweets we grab the text of the tweet and the id of the user.

The logical OR allows use to submit all 7 queries with one request to the API. This allows more requests to be made.

```
    # generates one request for several queries containing city + weather term
    query = '({0} {1} OR {0} {2} OR {0} {3} OR {0} {4} OR {0} {5} OR {0} {6} OR
{0} {7}) -is:retweet lang:en'.format(
            q, 'weather', 'rain', 'snow', 'sunny', 'hot', 'cold', 'warm')
    tweets = tweepy.Paginator(client.search_recent_tweets, query=query,
            tweet_fields=['context_annotations', 'author_id'],
max_results=100).flatten(limit=1000)
    for tweet in tweets:
        ids.append(tweet.author_id)
        corpus.append(tweet.text)
```

A score is now given using a BM25 function for each document using a generic query containing all our weather terms. I had planned to have a more sophisticated process for eliminating non-relevant documents, but I didn't find a good dataset to help me with it and had no time to create my own.

```
    # generate bm25 scores for relevance to weather terms and store in scores
    tokenized_corpus = [doc.split(" ") for doc in corpus]
    bm25 = BM25Okapi(tokenized_corpus)
    bm25_query = "{} weather rain snow sunny hot cold warm"
    tokenized_query = bm25_query.split(" ")
    doc_scores = bm25.get_scores(tokenized_query)
    scores_temp = pd.concat([pd.Series(ids, name='id'), pd.Series(corpus,
name='document'), pd.Series(doc_scores, name='score')], axis=1)
    scores = pd.concat([scores, scores_temp])
```

Duplicate tweets are dropped and so are documents that were given a score of 0 for relevance.

```
    # don't want duplicate tweets
    scores = scores.drop_duplicates()

    # remove documents(tweets) with 0 relevance scores
```

```
    scores = scores[scores['score'] > 0]
    scores.reset_index(drop=True, inplace=True)
```

With the documents we have deemed relevant, we now perform our sentiment analysis. This analysis was accomplished using the VADER sentiment analysis library. I would have liked to train my own model, but as with the BM25 function, I didn't have a good dataset and had little time and manpower to create my own. VADER seems to work well regardless. The largest issue was trying to filter out irrelevant tweets.

We create the data store for sentiment (SentVals), containing a positive, neutral, and negative value corresponding to the percentage of tweets given those classifications, as well as the average compound sentiment.

```
    # get sentiment values for each document
    sentiment = SentimentIntensityAnalyzer()
    sentiments = pd.DataFrame(columns=["id", "document", "neg", "neu", "pos",
"compound"])

    for i in range(scores.shape[0]):
            sent = sentiment.polarity_scores(scores['document'][i])
            sentiments.loc[len(sentiments.index)] = [scores['id'][i],
scores['document'][i],
                    sent['neg'], sent['neu'], sent['pos'], sent['compound']]

    sents = sentiments
    sents.drop(columns=['document'])

    # so we only have one entry per user, average all sentiment outputs for each
individual user account
    sents = sents.groupby('id', as_index=False,
sort=False).mean(numeric_only=True)

    # initialize and assign sentiment outputs
    comp_mean = '%.3f' % np.mean(sents['compound'])
    comp_size = len(sents['compound'])
    pos = '%.1f' % (100 * sum(sents['compound'] > 0.1)/comp_size)
    neu = '%.1f' % (100 * sum(0.1 >= abs(sents['compound']))/comp_size)
    neg = '%.1f' % (100 * sum(sents['compound'] < -0.1)/comp_size)

    data.SentVals = dict({'names' : ['comp_mean', 'pos', 'neu', 'neg'],
                          'values': [comp_mean, pos, neu, neg]})
```

Now we tokenized each word in the corpus, remove stop words, and count each word in the resulting body. All words from the relevant Tweets, along with their counts are stored in TweetTerms.

```
    # borrow some stopwords from nltk library and add a few others giving problems
    stop_words = {"i", "me", "my", "myself", "we", "our", "ours", "ourselves",
"you", "your",
        "yours", "yourself", "yourselves", "he", "him", "his", "himself", "she",
"her", "hers",
```

```
        "herself", "it", "its", "itself", "they", "them", "their", "theirs",
"themselves", "what",
        "which", "who", "whom", "this", "that", "these", "those", "am", "is",
"are", "was", "were",
        "be", "been", "being", "have", "has", "had", "having", "do", "does",
"did", "doing", "a",
        "an", "the", "and", "but", "if", "or", "because", "as", "until", "while",
"of", "at", "by",
        "for", "with", "about", "against", "between", "into", "through", "during",
"before", "after",
        "above", "below", "to", "from", "up", "down", "in", "out", "on", "off",
"over", "under",
        "again", "further", "then", "once", "here", "there", "when", "where",
"why", "how", "all",
        "any", "both", "each", "few", "more", "most", "other", "some", "such",
"no", "nor", "not",
        "only", "own", "same", "so", "than", "too", "very", "s", "t", "can",
"will", "just", "don",
        "should", "now", 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
'l', 'm',
        'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'co',
'new',
        'york', 'los', 'angeles', 'chicago', 'washington', 'dc', 'san',
'francisco', 'boston',
        'dallas', 'houston', 'philadelphia', 'atlanta', 'miami', 'detroit',
'phoenix', 'seattle',
        'orlando', 'minneapolis', 'denver', 'cleveland', 'diego', 'portland',
'tampa', 'st',
        'louis', 'charlotte', 'salt', 'lake', 'sacramento', 'buffalo', 'alabama',
'al', 'alaska',
        'ak', 'arizona', 'az', 'arkansas', 'ar', 'california', 'ca', 'colorado',
'connecticut',
        'ct', 'delaware', 'de', 'florida', 'fl', 'georgia', 'ga', 'hawaii', 'hi',
'idaho', 'id',
        'illinois','indiana', 'in', 'iowa', 'ia', 'kansas', 'ks', 'kentucky',
'ky', 'louisiana',
        'la', 'maine', 'me', 'maryland', 'md', 'massachusetts', 'ma', 'michigan',
'mi', 'minnesota',
        'mn', 'mississippi', 'ms', 'missouri', 'mo', 'montana', 'mt', 'nebraska',
'ne', 'nevada',
        'nv', 'new', 'hampshire', 'nh', 'jersey', 'nj', 'mexico', 'nm', 'york',
'ny', 'north',
        'carolina', 'nc', 'dakota', 'nd', 'ohio', 'oh', 'oklahoma', 'ok',
'oregon', 'or',
        'pennsylvania', 'pa', 'rhode island', 'ri', 'south', 'sc', 'sd',
'tennessee', 'tn', 'texas',
        'tx', 'utah', 'ut', 'vermont', 'vt', 'virginia', 'va', 'wa', 'west', 'wv',
'wisconsin', 'wi',
        'wyoming', 'wy', 'https', 'weather', 'will', 'great', '', ' '}

    # gather all terms from all documents
    doc_terms = ' '.join(sentiments['document'])
    # split the values with non-alpha as delimiter
    tokens = re.split('[^a-zA-Z]', doc_terms)
```

```
    # convert each token into lowercase
    for i in range(len(tokens)):
        tokens[i] = tokens[i].lower()

    # use list comprehension to remove stopwords
    filtered_tweet_terms = [i for i in tokens if i not in stop_words]

    data.TweetTerms = []

    # get value counts to be used for bar chart
    tweet_terms = pd.Series(filtered_tweet_terms).value_counts()
    for name, weight in tweet_terms.items():
        data.TweetTerms.append({'name': name, 'weight' : weight})

    return render_template("index.html",CityName=CityName)
```

Now we provide routes for each of our data stores to be received by the front end

```
# Provide route for all data
@app.route("/get-wv-data",methods=["GET","POST"])
def returnWVData():
    g=data.WeatherVals

    return jsonify(g)

@app.route("/get-wo-data",methods=["GET","POST"])
def returnWOData():
    h=data.WeatherObs

    return jsonify(h)

#@app.route("/get-cl-data",methods=["GET","POST"])
#def returnCLData():
#    m=data.CloudLyr

#    return jsonify(m)

@app.route("/get-sv-data",methods=["GET","POST"])
def returnSVData():
    f=data.SentVals

    return jsonify(f)

@app.route("/get-tt-data",methods=["GET","POST"])
def returnTTData():
    n=data.TweetTerms

    return jsonify(n)
```

And we run our Flask app.

```
if __name__ == "__main__":
    app.run(debug=True)
```

index.html

The HTML and CSS is straightforward so I'll only explain the JavaScript code sections inside the html document, although there isn't much to explain.

First we include d3 libraries for data retrieval and visualizations.

```
src="https://d3js.org/d3.v4.js">
```

Then we read in the WeatherVals and SentVals data and organize it for rendering.

```
  // Fill in text data
  d3.json("/get-wv-data", function(data) {
    document.getElementById("weathervals").innerHTML = data.values.join("<br
class='largebr'>");
  })
  d3.json("/get-sv-data", function(data) {
    document.getElementById("sentvals").innerHTML =
      `Compound Sentiment: ${data.values[0]}<br class='largebr'>Positive:
${data.values[1]}%
      <br class='largebr'>Neutral: ${data.values[2]}%<br class='largebr'>Negative:
${data.values[3]}%`;
  })
```

Then we read in the WeatherObs and TweetTerms data and use d3 to design the bar charts displaying the data.

```
// Create bar chart 1
var margin  = {top: 40, right: 35, bottom: 40, left: 85},
    width   = 770 - margin.left - margin.right,
    height  = 600 - margin.top - margin.bottom,
    y       = d3.scaleBand().range([0,height]).padding(0.5),
    x       = d3.scaleLinear().range([0,width]);
var xAxis = d3.axisBottom(x)
  .tickFormat(function(d){
    return d;
  }).ticks(10)
var yAxis = d3.axisLeft(y);
var svg1 = d3.select("#graphDiv")
  .append("svg")
  .attr("width", width + margin.left + margin.right)
```

```
  .attr("height", height + margin.top + margin.bottom)
  .append("g")
  .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

// Parse the Data
d3.json("/get-wo-data", function(data) {

  // Add axes
x.domain([0, d3.max(data, function(d){
  return d.weight;
})]);

y.domain(data.map(function(d){
  return d.name;
}));

svg1.append("g")
  .attr("transform", "translate(0, " + height + ")")
  .call(xAxis)

svg1.append("g")
  .call(yAxis)

// Add title label
svg1.append("text").attr("x", (width / 2))
  .attr("y", 0 - (margin.top / 2))
  .attr("text-anchor", "middle")
  .style("font-size", "16px")
  .text("Weather Descriptions by Frequency");


svg1.selectAll(".bar")
  .data(data)
  .enter()
  .append("rect")
  .style("fill", "darkgreen")
  .attr("x", x(0))
  .attr("y", function(d) { return y(d.name); })
  .attr("width", function(d) { return x(d.weight); })
  .attr('height', y.bandwidth())
})

// Create bar chart 2
var margin  = {top: 40, right: 35, bottom: 40, left: 85},
    width   = 770 - margin.left - margin.right,
    height  = 600 - margin.top - margin.bottom,
    y       = d3.scaleBand().range([0,height]).padding(0.5),
    x       = d3.scaleLinear().range([0,width]);
var xAxis = d3.axisBottom(x)
.tickFormat(function(d){
return d;
}).ticks(10)
var yAxis = d3.axisLeft(y);
var svg2 = d3.select("#graphDiv2")
```

```
    .append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
    .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

  // Parse the Data
  d3.json("/get-tt-data", function(data) {

  console.log(data);

  // Get Top Elements
  data=data.slice(0, 25);
  console.log(data);

  // Add axes
  x.domain([0, d3.max(data, function(d){
    return d.weight;
  })]);

  y.domain(data.map(function(d){
    return d.name;
  }));

  svg2.append("g")
    .attr("transform", "translate(0, " + height + ")")
    .call(xAxis)

  svg2.append("g")
    .call(yAxis)

  // Add title label
  svg2.append("text").attr("x", (width / 2))
    .attr("y", 0 - (margin.top / 2))
    .attr("text-anchor", "middle")
    .style("font-size", "16px")
    .text("Most Frequent Tweet Terms");

  svg2.selectAll(".bar")
    .data(data)
    .enter()
    .append("rect")
    .style("fill", "darkgreen")
    .attr("class", "bar")
    .attr("x", x(0))
    .attr("y", function(d) { return y(d.name); })
    .attr("width", function(d) { return x(d.weight); })
    .attr('height', y.bandwidth())
  })
```

Workload Summary

Much of the time for this project was spent figuring out how to implement the front end, as I had no prior experience with that. I tried a more complex full stack approach using Django and ReactJS but the project ended up overly extensive for the simple single page application I was attempting to implement. I then tried Bokeh but had trouble customizing it to my liking. So I turned to Flask and D3.js.

The approximate time spent on this project is broken down below:

Task	**Time**
Initial planning and research for analysis	7 hours
Back end (app.py) coding	12 hours
Django and ReactJS trial	14 hours
Bokeh trial	6 hours
Flask and D3 implementation	11 hours
Additional fine tuning	6 hours
Total	**56 hours**

Installation and Running the Software

The project was created using:

Python 3.10.5

Node 19.2.0

NPM 8.9.13

For information on installing a Node Version Manager (NVM) as well as Node and NPM you can look here:

https://docs.npmjs.com/downloading-and-installing-node-js-and-npm

Keep in mind a Node version needs to be activated (using command nvm use ...) after installing, otherwise the npm command will not work.

To clone the repository run the following from the command line:

```
git clone https://github.com/josh-monto/weather_sent.git
```

A bearer token must be added to a 'bearer_token.txt' file in the root folder of this project to access the Twitter API. I have included a bearer token from a secondary developer account for project evaluation, but I will revoke it once evaluation is over. After that, a token can be generated once a Twitter developer account is created. Instructions here:

https://developer.twitter.com/en/docs/authentication/oauth-2-0/bearer-tokens

I didn't need a token for the NWS API.

From this point forward it is recommended to run a virtual environment (https://docs.python.org/3.10/library/venv.html). Flask will not function if it is not installed and run from a virtual environment

Now, with Node and NPM installed (and activated), repository cloned, and virtual environment activated, navigate to the project root folder in the command line and run the following commands to install d3 (must be version 4) and all Python packages:

```
npm install --save d3
pip install flask tweepy noaa_sdk vaderSentiment pandas rank_bm25
```

Now run the project:

```
python -m flask run
```

Once the message `Running on http://127.0.0.1:5000` appears, navigate to http://127.0.0.1:5000 in a browser window. The page may take up to 15 seconds to open.

Once the page opens, you can select a city from the dropdown menu and click Submit once (the Submit button does not currently deactivate while code runs). Wait up to 15 seconds for the page to update with information for the selected city.