Yihang Fu
yf329
yihang.fu@yale.edu

Assignment 3
BIS 634 Computational Methods for Informatics

2024/11/08

1 Exercise 1

1.1 Main part of the gradient descent algorithm

To find optimal values of a and b that minimize the error, we can implement a two-dimensional version of the gradient descent algorithm. In this context, optimal parameters refer to those that minimize the error returned by the provided API.

Estimating the Gradient

Since we do not know the specific function, we cannot directly calculate the partial derivatives. Therefore, we can estimate the gradient numerically. For parameters a and b, the partial derivatives can be approximated as follows:

$$\frac{\partial f}{\partial a} \approx \frac{f(a+\epsilon, b) - f(a,b)}{\epsilon}$$
$$\frac{\partial f}{\partial b} \approx \frac{f(a, b+\epsilon) - f(a,b)}{\epsilon}$$

where $f(a,b)$ represents the error obtained from the API, and ϵ is a small step size. I chose $\epsilon = 0.001$.

Numerical Choices

- **Initial Parameters:** I start with random initial values of a and b within the range $(0, 1)$, which allows exploration of different regions of the parameter space.
- **Learning Rate:** A learning rate $\eta = 0.01$ was selected. Actually, I tried the learning rate in the set 0.01, 0.005, 0.001 and fixed the starting points. I found that with a learning rate = 0.001, it is much slower for the algorithm to converge (over 500 iterations, about 30 minutes with 3 starting points). Also, I found that there is no quite difference on the final convergence results among the first two learning rates. It seems that both of them will be 'stuck' in the local minimum. Therefore, I chose $\eta = 0.01$, and try to find out the local/global minimum
- **Stopping Criteria:** The algorithm stops when the change in error between iterations is less than 10^{-5}, or after a maximum of 1000 iterations. This method ensures efficiency and relative accuracy at the same time.

Gradient Descent Algorithm Implementation (see the code snippet 1)

For each iteration of gradient descent, we:

1. Calculate the current error $f(a,b)$ from the API.
2. Compute the approximate gradients for a and b using the finite difference method.
3. Update a and b as follows:
$$a = a - \eta \cdot \frac{\partial f}{\partial a}$$
$$b = b - \eta \cdot \frac{\partial f}{\partial b}$$
4. Check the stopping criteria.

This approach ensures convergence toward a minimum error value while making efficient use of API calls.

```python
def f(a,b,h=1e-5):
    result = float(requests.get(f"http://ramcdougal.com/cgi-bin/error_function.py?a={a}&b={b}", headers={"User-Agent": "MyScript"}).text)
    return result
#Since we do not know the function, we cannot calculate the true gradient of the function, we have to do estimation.
def gradient(a,b,h=1e-3):
    df_da = (f(a + h, b) - f(a, b)) / h
    df_db = (f(a, b + h) - f(a, b)) / h
    return df_da, df_db

def gradient_descent(start_a, start_b, learning_rate=0.01, tolerance=1e-5, max_iterations=1000):
    a, b = start_a, start_b
    for iteration in range(max_iterations):
        df_da, df_db = gradient(a, b,h=1e-3)

        # Update parameters
        new_a = a - learning_rate * df_da
        new_b = b - learning_rate * df_db

        # Check for convergence-This is our 'stopping strategy' (the iteration number is actually another stopping)
        if abs(new_a - a) < tolerance and abs(new_b - b) < tolerance:
            print(f"Converged in {iteration} iterations.")
            break

        a, b = new_a, new_b
        #print(f"Iteration {iteration}: a = {a}, b = {b}, f(a, b) = {f(a, b)}")

    return a, b,f(a, b)
```

Listing 1: Gradient Descent implementation

Justification of Choices

The values for the learning rate, step size ϵ, and stopping criteria were selected to balance convergence speed and accuracy. A moderate learning rate and small step size provide stable convergence, while the stopping criteria prevent unnecessary API queries. These choices are reasonable for minimizing error while efficiently locating optimal parameter values.
A straightforward method to search for the local minimum and the global minimum is to set random starting points and repeat trials. However, it is very time-consuming (especially when we have to call the API iteratively) and therefore hard for us to tune the starting point. Inspired by the article[1], here is the simulated annealing method for gradient descent. The algorithm operates as follows:

1. Start with an initial solution, which could be a random point in the search space for parameters a and b.
2. Set an initial "temperature" T, which controls the probability of accepting worse solutions. At higher temperatures, the algorithm is more likely to accept worse solutions, allowing it to explore more of the search space.
3. For each iteration:
    - Generate a small random change in the current parameters a and b, creating a "neighbor" solution.
    - Evaluate the error at this new solution using the error function API.
    - Calculate the difference in error, $\Delta E = E_{\text{new}} - E_{\text{current}}$.
    - If $\Delta E < 0$, accept the new solution, as it decreases the error.
    - If $\Delta E \geq 0$, accept the new solution with probability $\exp(-\Delta E/T)$, allowing for occasional "uphill" moves that help escape local minima.
4. Gradually reduce the temperature T according to a cooling schedule, typically $T = T_0 \times \alpha^k$, where T_0 is the initial temperature, α is a cooling rate between 0 and 1, and k is the iteration count.
5. Terminate the algorithm when the temperature T is sufficiently low or after a maximum number of iterations.

With this method, we can quickly find the local/global minimum. The detailed implementation is shown in the code snippet.

```python
def simulated_annealing(initial_a, initial_b, temp=100, cooling_rate=0.95, min_temp=1e-3, max_iterations=1000, perturbation_scale=0.1):
    a, b = initial_a, initial_b
    current_value = f(a, b)
    best_a, best_b = a, b
    best_value = current_value

    iteration = 0
    while temp > min_temp and iteration < max_iterations:
        # new candidate solution
        new_a = a + random.uniform(-perturbation_scale, perturbation_scale)
        new_b = b + random.uniform(-perturbation_scale, perturbation_scale)

        # Ensure the new values for a and b are within the range (0, 1), I just use 0.00001 and 0.99999 as an estimation
        new_a = max(0.00001, min(0.99999, new_a))
        new_b = max(0.00001, min(0.99999, new_b))

        # Calculate the function value at the new candidate solution
        new_value = f(new_a, new_b)

        # Calculate the difference in function values (energy difference)
        delta_value = new_value - current_value

        # Acceptance criteria
        if delta_value < 0 or random.random() < np.exp(-delta_value / temp):
            # Accept the new solution
            a, b = new_a, new_b
            current_value = new_value

            # Update the best found solution if the new one is better
            if current_value < best_value:
                best_a, best_b = a, b
                best_value = current_value

        # 'Cool down' the temperature
        temp *= cooling_rate
        iteration += 1

        print(f"Iteration {iteration}: a={a:.4f}, b={b:.4f}, f(a, b)={current_value:.4f}, temp={temp:.4f}")

    return best_a, best_b, best_value
```

Listing 2: Simulated Annealing Gradient Descent

```python
initial_points = [(0.9, 0.1), (0.5, 0.5), (0.1, 0.9)]
results = []

for initial_a, initial_b in initial_points:
    a, b, min_f = gradient_descent(initial_a, initial_b)
    results.append((a, b, min_f))

# Determine which is local and which is global
results.sort(key=lambda x: x[2])  # Sort by function value to identify the minima
global_min = results[0]
local_min = results[1]

print(f"Global minimum found at a={global_min[0]}, b={global_min[1]}, with f(a, b)={global_min[2]}")
print(f"Local minimum found at a={local_min[0]}, b={local_min[1]}, with f(a, b)={local_min[2]}")
```

Listing 3: GD with selected starting point

With the trials, we can find that when a=0.7238289792688082, b=0.17130068184700786, f(a, b)=1.00043565366 (global minimum). To double-check the result, I went back to the conventional method, choosing several starting points (with reference to the result derived by the simulated annealing method), for example (0.1,0.9), (0.5,0.5),(0.9,0.1). We have the final result: Global minimum found at a=0.7116628873000056, b=0.1684408077999854, with f(a, b)=1.00000127902; Local minimum found at a=0.2152305954000199, b=0.6889933245000216, with f(a, b)=1.10000059203.

2 Exercise 2

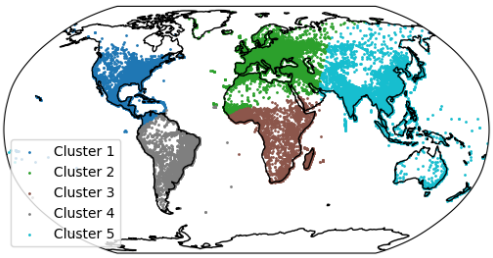
The data we use in the exercise come from SingleMaps which is licensed under CC BY 4.0. Here we acknowledge their great work contributing on this great dataset[1]
This code snippet shows the K-means algorithm modified from the one shown in class.

```
df = pd.read_csv('./simplemaps_worldcities_basicv1.77/worldcities.csv')

def lat_lon_to_cartesian(lat, lon):
    """Convert latitude and longitude to Cartesian coordinates."""
    lat_rad = np.radians(lat)
    lon_rad = np.radians(lon)
    x = np.cos(lat_rad) * np.cos(lon_rad)
    y = np.cos(lat_rad) * np.sin(lon_rad)
    z = np.sin(lat_rad)
    return np.array([x, y, z])

def cosine_similarity(vec1, vec2):
    """Calculate cosine similarity between two vectors."""
    return np.dot(vec1, vec2) / (np.linalg.norm(vec1) * np.linalg.norm(vec2))

# Convert all points to Cartesian coordinates
pts = np.array([lat_lon_to_cartesian(lat, lon) for lat, lon in zip(df['lat'], df['lng'])])


k = 15  #5, 7, or 15
max_iters = 100  # number of max iteration
centers = random.sample(list(pts), k)
old_cluster_ids, cluster_ids = None, [-1] * len(pts)

# Lloyd's algorithm for k-means clustering with cosine similarity
for iteration in range(max_iters):
    old_cluster_ids = cluster_ids.copy()
    cluster_ids = []
    for pt in pts:
        # Assign each point to the closest centroid based on cosine similarity
        similarities = [cosine_similarity(pt, center) for center in centers]
        max_cluster = np.argmax(similarities)
        cluster_ids.append(max_cluster)

    for i in range(k):
        cluster_pts = [pt for j, pt in enumerate(pts) if cluster_ids[j] == i]
        if cluster_pts:
            new_center = np.mean(cluster_pts, axis=0)
            centers[i] = new_center / np.linalg.norm(new_center)  # Normalize to unit vector

    # Break the loop if clusters do not change iteratively
    if cluster_ids == old_cluster_ids:
        print(f"Converged in {iteration + 1} iterations.")
        break
else:
    print(f"Reached maximum iterations: {max_iters}")

df['cluster'] = cluster_ids

# Plotting the clusters on a world map, here we reuse the visualization function provided
    before
def plot_clusters(df, k):
    fig, ax = plt.subplots(subplot_kw={"projection": ccrs.Robinson()})
    ax.set_global()
    ax.coastlines()
    for i in range(k):
        cluster_df = df[df['cluster'] == i]
        ax.scatter(
            cluster_df['lng'], cluster_df['lat'],
            s=1, transform=ccrs.PlateCarree(),
            label=f"Cluster {i+1}"
        )

    ax.legend()
    plt.show()

plot_clusters(df, k)
```

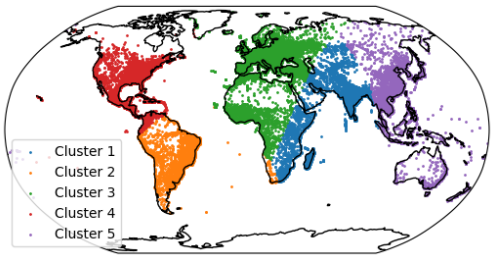
Listing 4: Modified K-means algorithm

[1] Simple Maps (https://simplemaps.com/data/world-cities), licensed under CC BY 4.0.

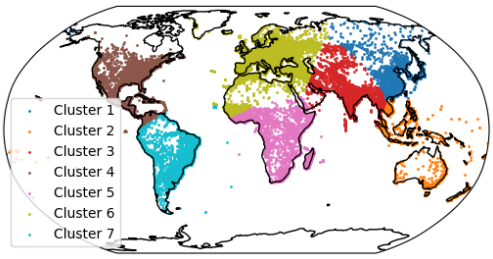
Here are the clustering results 1. As we can see, when k=5, the left-hand side clusters in a continent basis, while the right-hand side divide the eastern part of Africa with the Middle East. when k=7, the northern part of Africa is clustered with Europe in the left figure, and the southeastern Asia is clustered with Australia. While in the right hand side, the eastern Asia was separated from the west Asia; east Europe was separated from west Europe as well, which are quite reasonable. When k=15, the map is divided more concretely. For example, each continent can be divided into several parts. One explanation is that some data points in some regions are dense while in other regions are sparse.



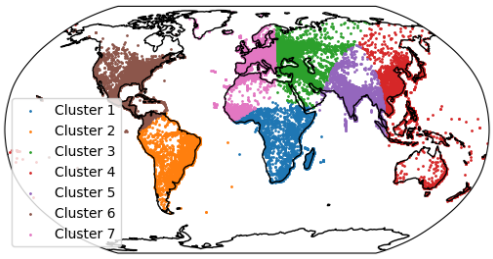
(a) K means result when k=5



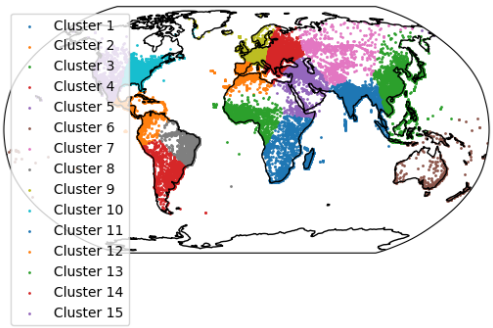
(b) K means result when k=5



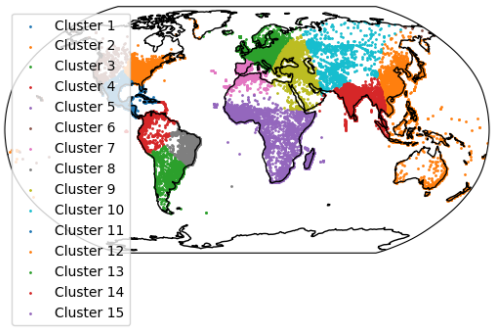
(c) K means result when k=7



(d) K means result when k=7



(e) K means result when k=15



(f) K means result when k=15

Figure 1: Overall caption for all pairs of figures

3 Exercise 3

```python
import numpy as np
import matplotlib.pyplot as plt

def euler_sir(S0, I0, R0, beta, gamma, Tmax, dt=0.1):
    # Initialize parameters
    N = S0 + I0 + R0
    S, I, R = S0, I0, R0
    t = 0
    times = [t]
    S_values, I_values, R_values = [S], [I], [R]

    while t < Tmax:
        # Euler method update
        dS = -beta * S * I / N
        dI = beta * S * I / N - gamma * I
        dR = gamma * I

        S += dS * dt
        I += dI * dt
        R += dR * dt

        # Update time
        t += dt
        times.append(t)
        S_values.append(S)
        I_values.append(I)
        R_values.append(R)

    return np.array(times), np.array(S_values), np.array(I_values), np.array(R_values)
```

Listing 5: Euler Method

As the question description introduced, we initiate the relative parameters as below. We got the result - Disease runs its course by time: 29.900000000000155, visualized in 2.

```python
#parameters
N = 137000
S0 = N - 1  # Initially susceptible
I0 = 1      # Initially infected
R0 = 0      # Initially recovered
beta = 2    # Infection rate
gamma = 1   # Recovery rate
Tmax = 100  # Sufficiently large time to let the epidemic run its course

# Run the simulation
times, S_values, I_values, R_values = euler_sir(S0, I0, R0, beta, gamma, Tmax)

# Find the time point where I < 1
for i, I in enumerate(I_values):
    if I < 1:
        print(f"Disease runs its course by time: {times[i]}")
        break

# Plotting I(t)
plt.plot(times, I_values, label="Infected I(t)")
plt.axhline(1, color='r', linestyle='--', label='Infected = 1')
plt.xlabel("Time")
plt.ylabel("Infected Individuals")
plt.title("Epidemic Progression in New Haven")
plt.legend()
plt.show()
```

Listing 6: New Haven Epidemic simulation

To find the peak infection time and maximum number of infections, we can simply use the built-in function in python to search the maximum element in an array, as shown in the code snippet below. The result - Peak time: 12.199999999999973; Maximum number of infected individuals: 21525.699125405023

```python
max_infected = max(I_values)
peak_time = times[np.argmax(I_values)]
print(f"Peak time: {peak_time}")
print(f"Maximum number of infected individuals: {max_infected}")
```

Since we do not know the exact parameter values, we vary both beta and gamma values in a certain acceptable range. Then, we visualize the values in heatmaps.

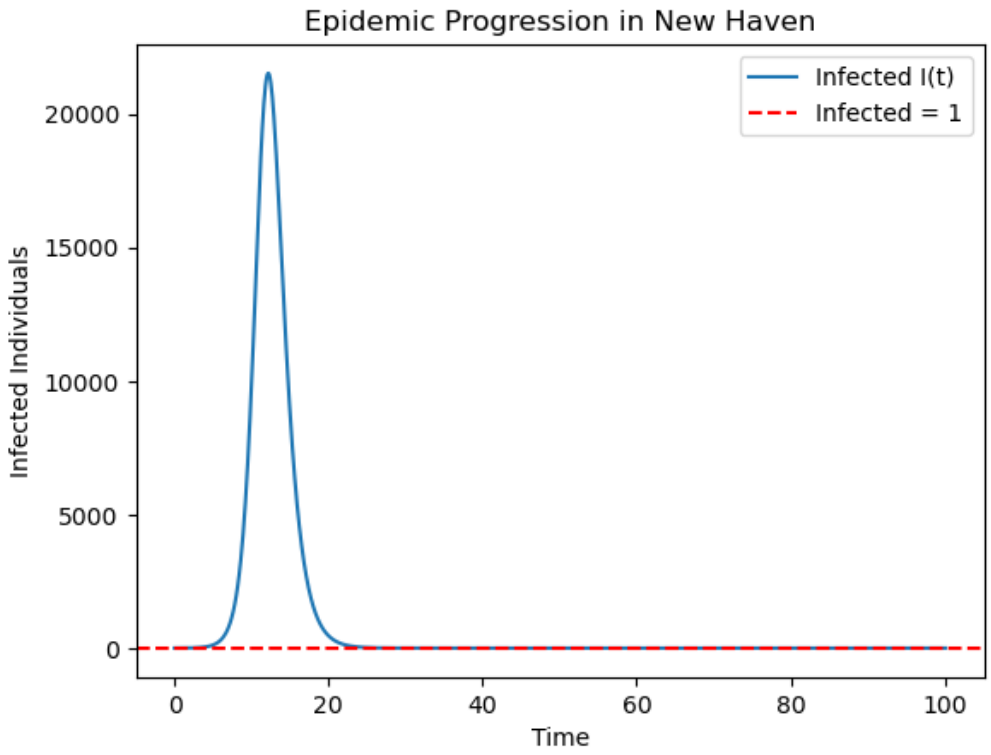


Figure 2: Epidemic Progression in New Haven

```python
import pandas as pd
import seaborn as sns

# Define ranges for beta and gamma
beta_values = np.linspace(1.5, 2.5, 10)
gamma_values = np.linspace(0.5, 1.5, 10)

peak_infection_times = []
peak_infection_counts = []

# Calculate peak infection time and count for each combination of beta and gamma
for beta in beta_values:
    row_times = []
    row_counts = []
    for gamma in gamma_values:
        times, S_values, I_values, R_values = euler_sir(S0, I0, R0, beta, gamma, Tmax)
        max_infected = max(I_values)
        peak_time = times[np.argmax(I_values)]
        row_times.append(peak_time)
        row_counts.append(max_infected)
    peak_infection_times.append(row_times)
    peak_infection_counts.append(row_counts)

# Convert results to DataFrames for plotting
df_times = pd.DataFrame(peak_infection_times, index=beta_values, columns=gamma_values)
df_counts = pd.DataFrame(peak_infection_counts, index=beta_values, columns=gamma_values)


### I asked CHATGPT to help me visualize the heatmap.
# #key assistance from CHATGPT: I want to change the font in each grid and keep the numbers in two-decimal.
plt.figure(figsize=(12, 5))

# Peak infection time heatmap
plt.subplot(1, 2, 1)
sns.heatmap(
    df_times,
    annot=True,
    fmt=".2f",
    cmap="YlGnBu",
    xticklabels=[f"{x:.2f}" for x in gamma_values],
    yticklabels=[f"{y:.2f}" for y in beta_values],
```

```
    annot_kws={"size": 8}  # Set font size for annotations
)
plt.title("Peak Infection Time (Heatmap)")
plt.xlabel("Gamma")
plt.ylabel("Beta")

# Peak infection count heatmap
plt.subplot(1, 2, 2)
sns.heatmap(
    df_counts,
    annot=True,
    fmt=".2f",
    cmap="YlOrRd",
    xticklabels=[f"{x:.2f}" for x in gamma_values],
    yticklabels=[f"{y:.2f}" for y in beta_values],
    annot_kws={"size": 6}  # Set font size for annotations
)
plt.title("Peak Infection Count (Heatmap)")
plt.xlabel("Gamma")
plt.ylabel("Beta")

plt.tight_layout()
plt.show()
```

Listing 7: Vary parameters and visualize heatmap

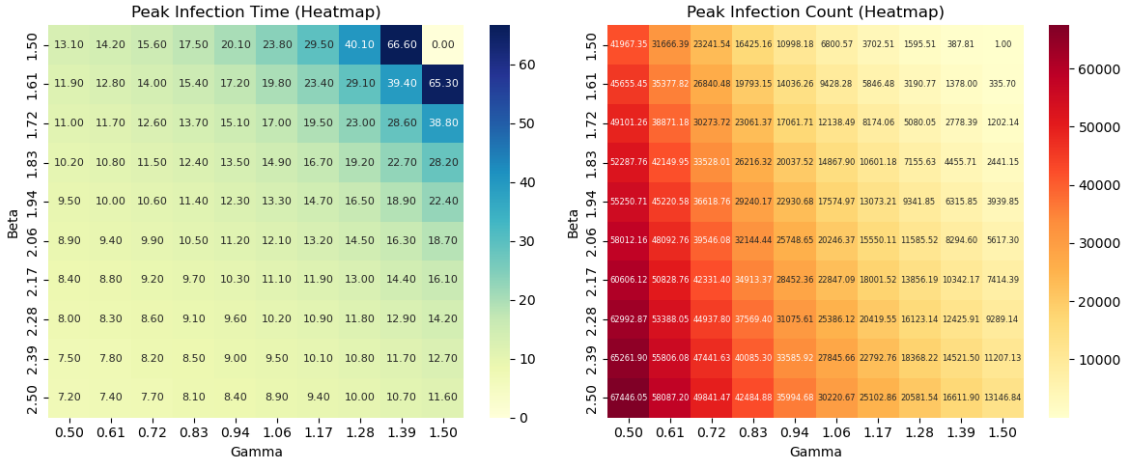


Figure 3: Heatmap result for varying parameters

4 Exercise 4

In this provided Flask example shown in 7, `server.py` is the main script that initializes the server and defines the routes. The root route ("/") renders `index.html`, which provides a form for users to input text for analysis. When the form is submitted, it sends a POST request to the "/analyze" route, which is also defined in `server.py`. This route retrieves the user's text, counts each character's occurrences using the `Counter` class, and prepares a formatted result. Then, `server.py` renders `analyze.html`, passing the original text and character count analysis as template variables. The `analyze.html` file displays the input text and the analysis result with a simple layout. Each file has a distinct role: `index.html` gathers user input, `server.py` processes the input, and `analyze.html` presents the output, creating a workflow in the application.

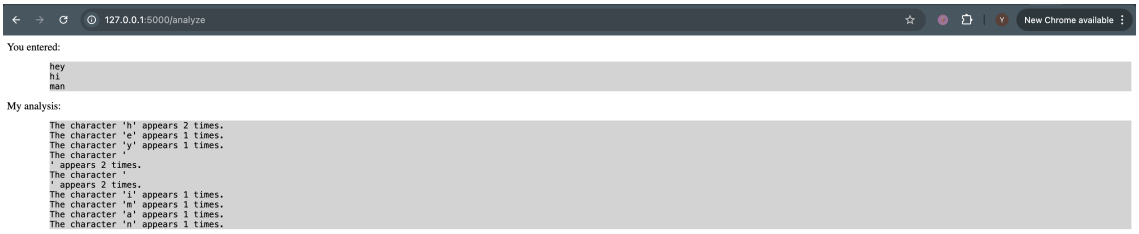


Figure 4: Flask Example demo

In our designed flask exercise, I do not select the data for my final project. Instead, I found CSV data from Johns Hopkins Corona Virus Resource Center[2] This data contains the time-series data of people infected by Covid-19 in each state in the US. In this exercise, the input is the name of a certain state in US and the output will be a time-series plot together with the peak number of infected population and the corresponding peak time. Also, we add a future pandemic prediction feature using a Linear Regression API. Since we simply call max() built-in function in Python, we can determine our peak result is correct. Since the cases report stopped by 2023.3.9, we cannot validate the prediction result. The code snippet below is my `server.py` file. `result.html` file and the `index.html`file.

```python
from flask import Flask, request, render_template
import pandas as pd
import matplotlib.pyplot as plt
from io import BytesIO
import base64
import matplotlib
from sklearn.linear_model import LinearRegression
import numpy as np

# Use non-GUI backend for Matplotlib
#This is under help of CHATGPT, I at first cannot show the visualization in the window and then I learned that I need to use the non-interactive backend.
matplotlib.use('Agg')

app = Flask(__name__)

# Load the data once when the server starts
try:
    data = pd.read_csv('time_series_covid19_confirmed_US.csv')
except Exception as e:
    print("Error loading data:", e)

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/result', methods=['POST'])
def result():
    state_name = request.form['state']
    state_data = data[data['Province_State'].str.lower() == state_name.lower()]

    if state_data.empty:
        result_text = f"State '{state_name}' not found in the dataset."
        return render_template('result.html', result=result_text)

    # Sum up all counties to get state-level data
    state_timeseries = state_data.iloc[:, 11:].sum(axis=0)
```

[2](https://github.com/CSSEGISandData/COVID-19), licensed under CC BY 4.0.

```
    state_timeseries.index = pd.to_datetime(state_timeseries.index, format='%m/%d/%y')

    # Prepare data for linear regression
    days = np.arange(len(state_timeseries)).reshape(-1, 1)  # Day indices as X
    cases = state_timeseries.values  # Case counts as y

    # Simple linear regression model
    model = LinearRegression()
    model.fit(days, cases)

    # Predict the next day's case count
    next_day = np.array([[len(days)]])  # The day after the last day in the dataset
    next_day_prediction = model.predict(next_day)[0]  # Predicted case count for the next
    day

    # Get the peak information
    peak_cases = state_timeseries.max()
    peak_date = state_timeseries.idxmax()

    # Create a result message
    result_text = (
        f"The peak for {state_name} was on {peak_date.date()} with {peak_cases} cases. "
        f"The predicted cases for the ({(state_timeseries.index[-1] + pd.Timedelta(days=1)).
    date()}) "
        f"is approximately {int(next_day_prediction)} cases."
    )

    # Plot the data with prediction
    fig, ax = plt.subplots()
    ax.plot(state_timeseries.index, cases, label='Actual Cases')
    ax.plot(state_timeseries.index[-1] + pd.Timedelta(days=1), next_day_prediction, 'ro',
    label='Predicted Next Day')
    ax.set_title(f"COVID-19 Cases Time Series for {state_name}")
    ax.set_xlabel("Date")
    ax.set_ylabel("Confirmed Cases")
    ax.legend()

    # Save the plot to a BytesIO object and encode it in base64,
    #this is under help of CHATGPT. I learned to use base64 to encode the image and decode
    to show the image in the window.
    img = BytesIO()
    plt.savefig(img, format='png')
    img.seek(0)
    plot_url = base64.b64encode(img.getvalue()).decode('utf8')
    plt.close(fig)

    return render_template('result.html', result=result_text, plot_url=plot_url, state_name=
    state_name)

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

Listing 8: server.py

```
<!DOCTYPE html>
<html>
<head>
    <title>COVID-19 Peak and Prediction</title>
    <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='style.css')
    }}">
</head>
<body>
    <h2>COVID-19 Peak and Prediction</h2>
    <p>{{ result }}</p>
    <img src="{{ plot_url }}" alt="COVID-19 Time Series Plot with
    Prediction">
    <a href="/">Search Again</a>
</body>
</html>
```

Listing 9: result.html

```
<!DOCTYPE html>
<html>
<head>
    <title>COVID-19 Peak Finder</title>
    <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='style.css')
    }}">
```

```
</head>
<body>
    <h2>Find the Peak Date of COVID-19 Cases for a U.S. State</h2>
    <form action="/result" method="POST">
        <label for="state">Enter State Name:</label>
        <input type="text" id="state" name="state" required>
        <button type="submit">Submit</button>
    </form>
</body>
</html>
```

Listing 10: index.html

This simple application has several features:

1. An error report and error handling.
2. layout design using CSS
3. relevant static image (after you input a legal state name, it will return a time-series plot)
4. An image generated in response to the input. (We add the data point based on the Linear regression prediction)
5. We integrate a Linear Regression API to make a simple prediction
6. multiple analysis: We have a simple time-series visualization, maximum peak report and a linear regression for future pandemic prediction.

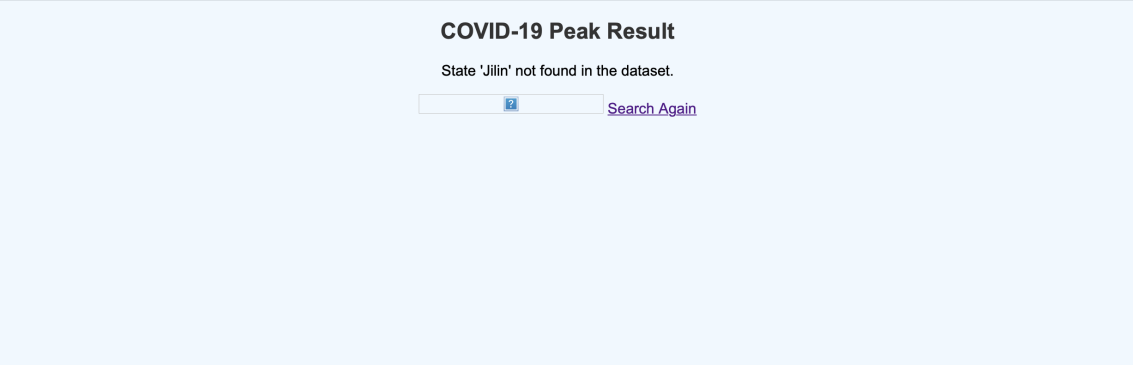


Figure 5: Flask Error report

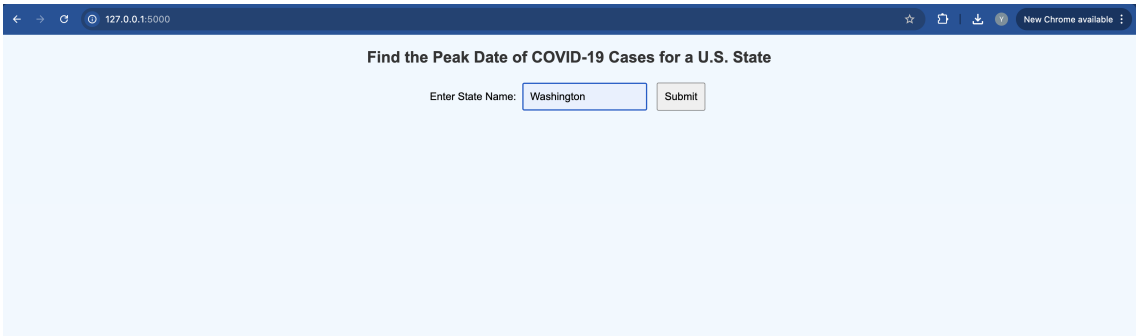


Figure 6: Flask input demo

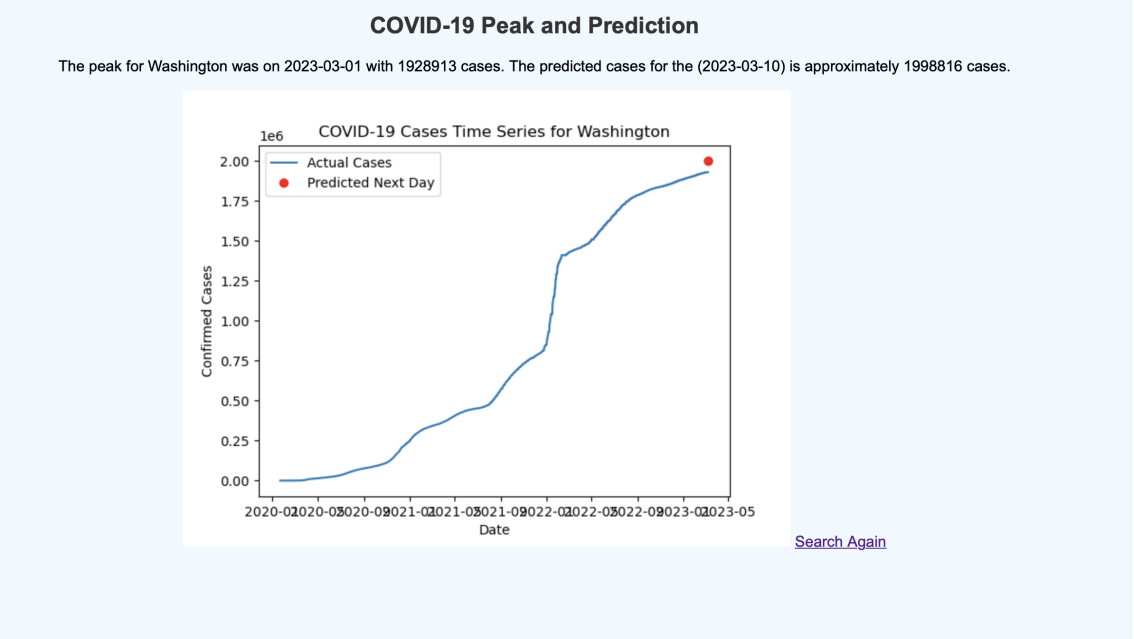


Figure 7: Flask API demo

References

[1] Zhicheng Cai. 2021. Sa-gd: Improved gradient descent learning strategy with simulated annealing. *arXiv preprint arXiv:2107.07558* (2021).