Optimizing Cardiac Emergency Responses in NYC: A Study on Volunteer Distribution and Its Impact on Survival Rates

Julia VanPutte (JHV42)

October 21, 2023

Abstract

Out-of-hospital cardiac arrest (OHCA) is a significant health challenge, and ensuring timely response to OHCAs is critical. Combining ambulance services with an app-alerted volunteer system can enhance response times. This study aims to determine the optimal number and distribution of volunteers across NYC to improve survival rates.

This report first analyzes the number of volunteers needed in NYC to improve survival rates. Volunteer presence across New York City is modeled as a Poisson point process. Here, analysis is restricted to a set number of 1163 locations across NYC, assuming that in each location the volunteer density is proportional to the population density in that location. Through this modeling, the response time distribution of the nearest volunteer is found. Then, using the de Maio survival function, response times are related to patient survival. It is shown that to maximize patient survival, it is ideal to maximize the number of volunteers across NYC.

Then, the question is raised of the optimal distribution of volunteers. Assuming there are 5,000 volunteers currently, and can recruit and additional 7,000 volunteers, we must decide the locations those new volunteers NYC. This will determine recruiting efforts. Incorporating optimization techniques, the ideal volunteer location distribution is found. By optimizing the volunteer distribution, this can determine where recruiting efforts for additional volunteers should be focused.

These findings present a blueprint for the recruitment and deployment of volunteers in New York City. The insights highlight the significance of the volunteers' spatial distribution, indicating city regions where recruitment would yield maximum impact on survival. This analysis underscores the potential to elevate survival rates from OHCAs through volunteers in addition to traditional systems like ambulances.

1 Introduction

As the most densely populated city in the United States, New York City faces unique demands when it comes to its healthcare and emergency response systems. Ensuring rapid and efficient response times to Out-of-hospital cardiac arrests (OHCAs) across the metropolitan area is critical. OHCA is a major cause of mortality across the world. The probability of survival significantly improves if patients have early access to treatment in the form of CPR.

A new way to improve the time until the patient received CPR is through a network of trained volunteers who are alerted through an app when a OHCA happens nearby. However, the success of this supplementary system hinges on two fundamental questions: How many volunteers are essential for a significant impact on survival rates in a place as vast as NYC? And more crucially, how should these volunteers be distributed across the city to ensure maximum efficiency?

In this report, a model for the impact of dispatching volunteers is developed. A key metric considered is the probability of patient survival, which is determined based on the response time. A key goal is to determine how the number of volunteers in NYC, and the location in which they are recruited, affect overall patient survival. Here, this analysis is critical for enhancing OHCA survival rates in NYC.

2 Analysis

The analysis is twofold. First, the distribution of volunteers across the city is modeled as a Poisson point process. The affect of the number of volunteers in NYC on the survival rates is determined. Second, the optimal volunteer distribution across NYC is found. This determines locations where it is crucial to recruit additional volunteers.

2.1 Modeling Volunteer Response

2.1.1 Modeling Assumptions

In order to model Volunteer Response, assumptions must be made on the model.

- Volunteers are alerted exactly 3 minutes after the OHCA.
- Any given volunteer is available and willing to accept this alert with a probability 0.3 (30% Acceptance).
- NYC is restricted to those locations we used in the solutions to HW 1.
- When an OHCA arises, the probability that it came from Location i is proportional to the population in Location i.
- The probability of survival is dependent on the time that the volunteer arrives. The function used for this is the de Maio survival function. $s(t) = (1 + e^{0.697+0.262t})^{-1}$.
- In each location the volunteer density is proportional to the population density (people per square kilometer) in that location.
- Volunteers at each location are uniformly distributed in a circle with radius proportional to location area around each location. Some volunteers will be shown as not on land, but this is a simplifying assumption.
- We have 5,000 volunteers right now, and can recruit more.
- Ambulances arrive at a constant time of 9 minutes. An ambulance will always arrive at the OHCA in 9 minutes. This is a simplifying assumption but appropriate for analysis of volunteers.
- Volunteers walk in straight lines at a constant speed of 6km/hr
- There are 177/2 calls per hour, and these are distributed according to population at a Poisson point process. However, we do not need to model this process since on average each volunteer is alerted about once per year, due to location, even through there are many OHCAs across the city.

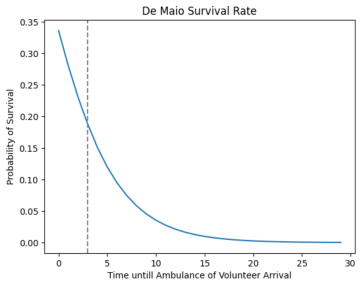


Figure 1: de Maio survival rate

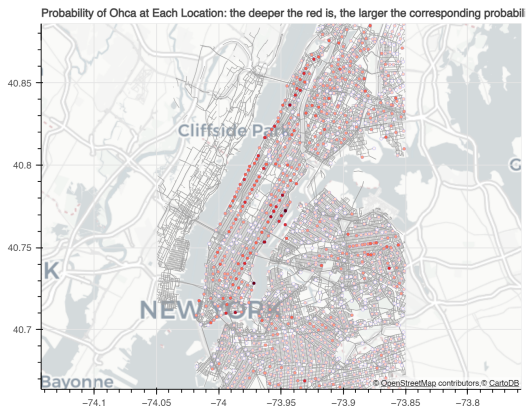


Figure 2: Probability of OHCA in each location. The deeper the red, the larger the corresponding probability of OHCA

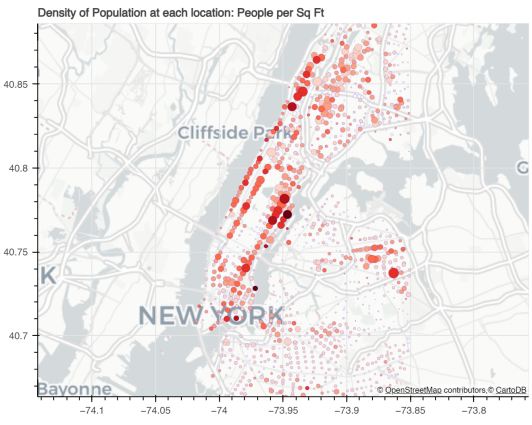


Figure 3: Density of Population at each location in People per Sq Km. The size of the node indicates area, and the color indicates population at the node.

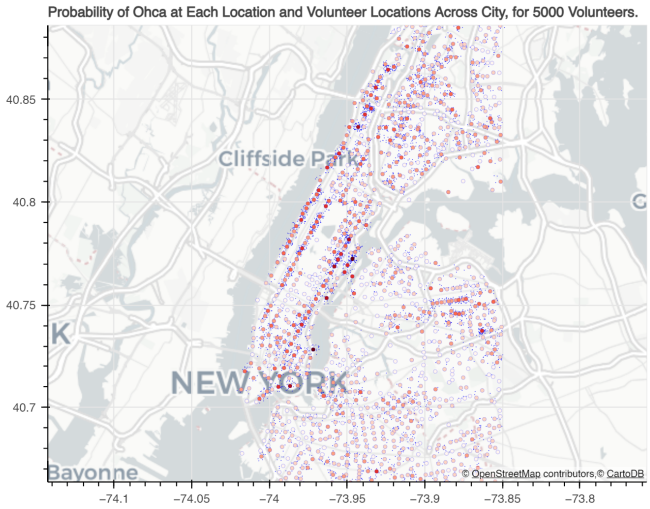


Figure 4: Distribution of 5000 volunteers across NYC

2.1.2 Volunteer Poisson Point Process

Available volunteers are distributed throughout NYC according to a spacial Poisson point process. It is assumed that available volunteers does not depend on number of calls since volunteers are only alerted about once per year and OHCAs are not very common events. When the number of volunteers n is large, a Poisson point process model is appropriate. If we have N volunteers across the city, we have 0.3N available volunteers. This will converge to a Poisson process of mean λ where $\lim_{n\longrightarrow\infty}\sum_{i=1}^{n}.3v_i(l) = \lambda(l)$ where .3 is the probability that volunteer i is available, and v_i is the location distribution of volunteer i in the city based on location l. Then set $\lambda = n\alpha v$, with $\alpha = 0.3$ for all volunteers across the city. The location probability distribution will be the same probability distribution as OHCAs, proportional to population in each location (node here).

Then, the distribution of response times t of the closest volunteer needs to be found. t will be the time until the first volunteer or ambulance arrives. So, $t = min(t_{vol}, t_{amb})$ We know $t_{amb} = 9$ So, to get t_{vol}, we have t_{vol} =response delay + walking time We know responsedelay = 3 min. So, to get t_{walk}, we can define the region around the OHCA i as region j. Lets define this region to be l. Then, we know that this is also a Poisson point process with mean $\lambda_l = n\alpha v(l)$. From study of Poisson process, we know that $P(T_{walk} \leq t) = 1 - e^{-\lambda l}$. This will give us the probability that the response time of the closest volunteer is less than t. We then can set λ_l to be the constant available volunteer density in the area. So, $P(T_{walk} \leq t) = 1 - e^{-\lambda_l \pi d_t^2}$ where $d_t = s(t - T_{walk})$ and s is volunteer walking speed in km/minute. We assume speed is 6km/hr or 0.1km/min.

Since we know that volunteer response delay is 3 minutes, We then get, for the probability that there is $P(T_{walk} \leq t) = 1 - e^{-\lambda_l \pi s^2 (t-3)^2}$

And here, λ_l = available volunteer density in the area, in volunteers per km. So, λ_l = (Total Volunteers in City) / (Total population of City) * (Population Density in Area) and Population Density in area = (Total Population in area) / (Total Area). Define p_l to be population in area l, a_l to be the area in km^2 of area l, N total volunteers, and P to be total population in the city, and A to be the area of the city in km^2. $\lambda_l = .3Np_l/Pa_l$

$$P(T_{walk_l} \leq t) = 1 - e^{-\frac{.3Np_l}{Pa_l}\pi s^2(t-3)^2}$$

Then, to get the CDF of all volunteer response times, We have a probability that a call arises in a given region, P_l, and $\sum_l P_l = 1$.

$$P(T_v \leq t) = \int_l P_l 1 - e^{-\frac{.3Np_l}{Pa_l}\pi s^2(t-3)^2}$$

The PDF of this function is

$$f(t) = \int_l P_l * \frac{2\pi.3Np_l s^2(t-3)\mathrm{e}^{-\frac{\pi.3Np_l s^2 ft(t-3)^2}{Pa_l}}}{Pa_l}$$

Then, the CDF and PDF of response times for different number of volunteers can be plotted. We can use the distribution of response times to get the average survival rate. $E[s(t)] = \sum_t f(t)s(t)$ Mean response times are shown for a system with no ambulances and a system with ambulances. These graphs are quite similar indicating that

even without ambulances, volunteers can help people with cardiac arrests greatly. This underscores the importance of the volunteer response app. Here we also show the incremental benefit of additional volunteers. The incremental benefit decreases as volunteers increase, but benefit is still positive for recruiting additional volunteers.

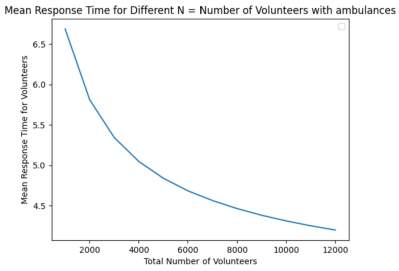


Figure 5: Mean Response time with Ambulances

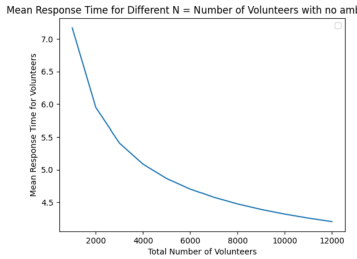


Figure 6: Mean Response time with no Ambulances

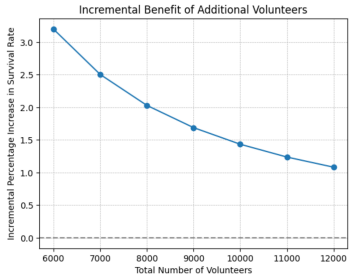


Figure 7: Incremental Benefit of Volunteers

Now, average survival rate is shown. It is clear that for the best increase in survival rate, a maximum possible number of volunteers should be recruited. Also, a bound for the maximum survival rate, $s(3)$ is shown. When survival was calculated for 1,000,000 volunteers, the survival rate was very close the the maximum survival rate. This indicates the analysis is correct. (Mean Survival Rate for 1,000,000 volunteers = 0.1825 ; Max Survival Rate 0.1877)

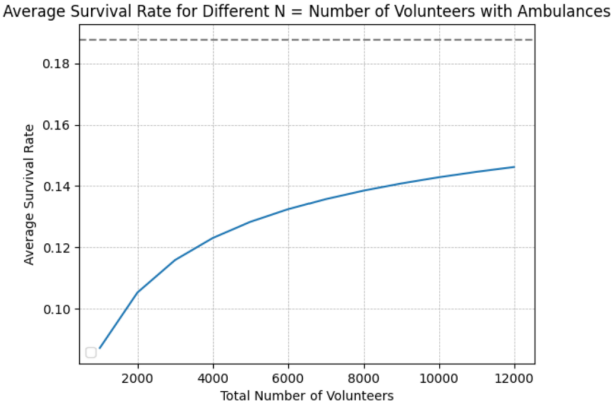


Figure 8: Average Survival Rate with Ambulances

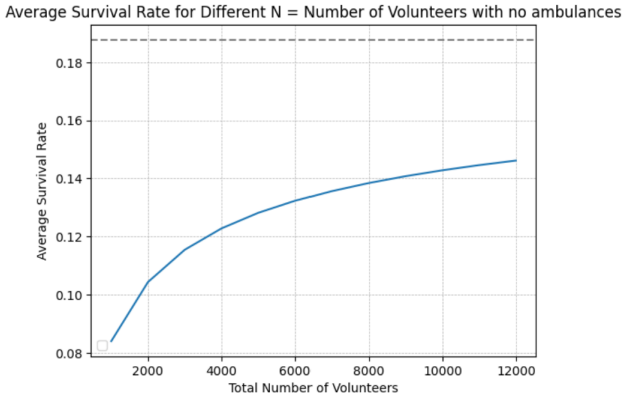


Figure 9: Average Survival Rate with no Ambulances

As shown in the graphs, a maximum number of volunteers will result in the most increase in survival rate. NYC should attempt to recruit all 12,000 volunteers.

2.2 Volunteer Distribution

Given the initial 5,000 volunteers in New York City (NYC), distributed in alignment with the population density of various locations, an additional 7,000 volunteers are intended to be added. This gives a total of 12,000 volunteers. The primary aim is to strategically place these volunteers across NYC to achieve the highest possible increase in survival rates.

When distributing these volunteers, we consider the intensity associated with the Poisson point process. This will guide recruiting efforts. Imagine this intensity as a vector that varies by location and integrates to 12,000. We desire to maximize this intensity for better survival rates.

The probability of death is intrinsically linked to the response time of volunteers. This is mathematically represented as: $p = f(T_v) \in [0, 1]$ where p is probability of death and T_v is the response time of the volunteer. The death probability reduces to its minimum when $T_v = 3$ since that represents the fastest response time by a volunteer. This minimum death probability is defined as d_{min}. Conversely, when the response time is infinite , the death probability is maximum, denoted by d_{max}. Since the response time is dependent on the probability of OHCA and the location of volunteers. We know that $T_v = f(v)$ where v is the distribution of volunteers across the city. The death probability as a function of v, the volunteer distribution, is $d(v) = E[f(T_v)] = \int_0^1 P(f(T_v) > t)du = d_{min} + \int_{d_{min}}^{d_{max}} P(f(T_v) > t)du$. We will assume that the first 5,000 volunteers are distributed according to the population across the city. These volunteers will be distributed according to a spacial Poisson process. For the

next 7,000 volunteers, we will allocate each volunteer to a location using a greedy approach. Volunteers are added to locations where they would yield the highest marginal reduction in death rate.

A greedy algorithm is implemented in the code in the appendix. Using this algorithm, we get the optimal allocation of volunteers across the city. The survival rate for proportional allocation of 12,000 volunteers across the city was about 0.145. The survival Rate for Optimal Allocation with 12,000 Volunteers is much better, at 0.1871. Comparing this to the maximum survival rate, which is 0.1877, this is very close. With an optimal allocation of 12,000 volunteers the survival rate can be almost maximized.

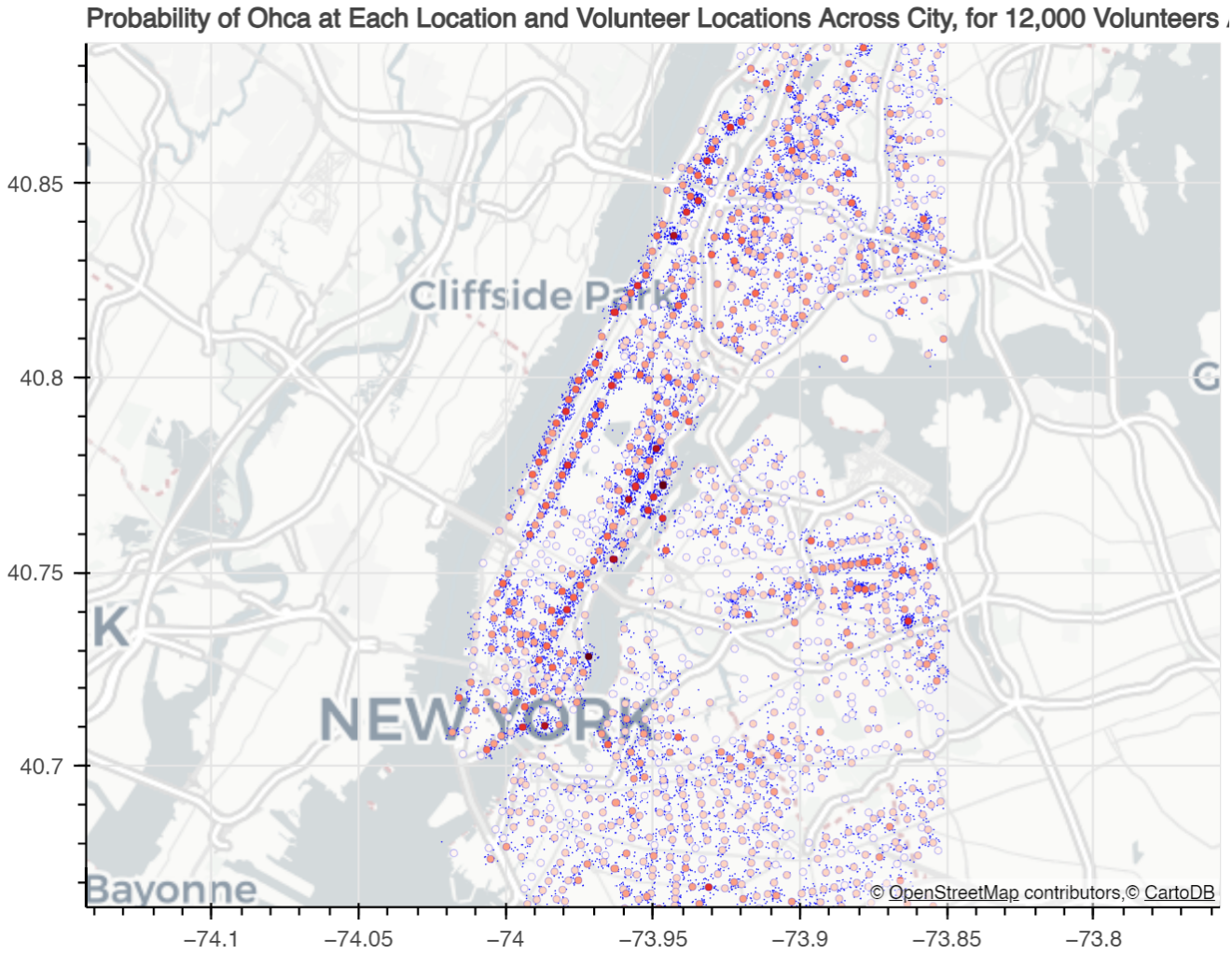


Figure 10: Optimal Distribution of 12,000 volunteers across NYC

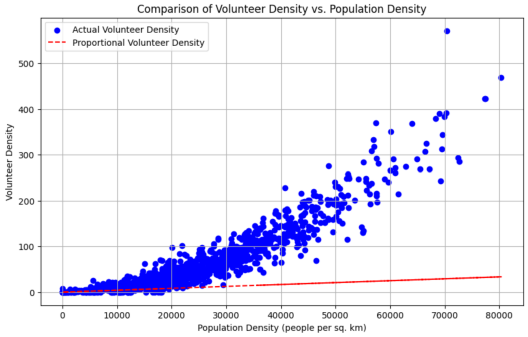


Figure 11: Comparison of Volunteer Density and Population Density

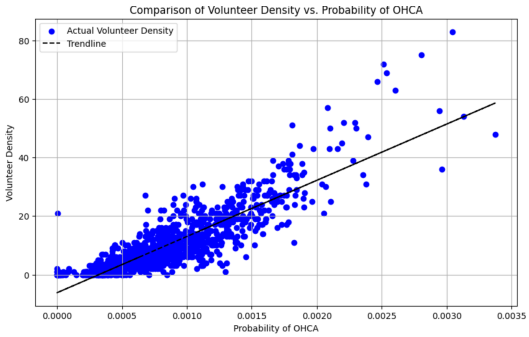


Figure 12: Comparison of Volunteer Density and Probability OHCA

From these figures, we see it is optimal to recruit volunteers at locations where the probability of OHCA is

high. This algorithm in the appendix can be used to determine the optimal distribution of volunteers, which will determine recruiting efforts across the city. This approach provides a systematic way to decide where the additional 7,000 volunteers should be placed to achieve the most substantial increase in survival rates, thereby optimizing the overall system and potentially saving more lives.

2.3 Discussion on 3-D Modeling

In this modeling, the model does not explicitly consider the fact that cities are 3-dimensional with high rise buildings. High-rise buildings in cities introduce several challenges when it comes to effectively responding to out-of-hospital cardiac arrests (OHCAs). Mapping systems are two-dimensional, and if the app alert does not specify the floor the volunteer could have trouble finding the patient. Entering a high-rise, especially secured residential or commercial buildings, often requires navigating through security gates, reception areas, or even locked entrances, causing delays. In very tall buildings, taking the stairs is often impractical, especially when every second counts. Reliability on elevators, which might not be immediately available, can delay response times. Even if a volunteer reaches the correct floor, finding the specific apartment or office of the individual in distress can be challenging, especially in buildings with complex layouts.

In order to extend the model to take a 3-D city into account. We could add an average wait time for elevators if the person is in a high rise. This could be based on the height of the building. Also, using data on the building type, we can estimate the average time taken to enter the building. Also, we could break each high rise into layers, where floors 1-10 are layer 1, floors 11-20 are layer 2, etc. Then, volunteers could be dispatched not just based on location but also based on which layer they can most quickly access. Further, if this system was in practice for a long time, the system could maintain a building access database for the most frequent buildings, helping volunteers to understand security protocols, access points, and key contacts. Further, incorporating detailed indoor instructions for frequently accessed buildings would be helpful. Also, alerting the volunteer on which floor the patient is on would help decrease response time. Incorporating these suggestions would require collaboration with building managements, investment in technology, and training for volunteers. However, by making the model more accurate for 3-D cities, the effectiveness of volunteer responses for OHCAs can be enhanced.

3 Conclusion

In addressing the critical matter of out-of-hospital cardiac arrests (OHCAs) within the vast expanse of New York City, this study offers a comprehensive examination of volunteer deployment strategies, population density considerations, and the incorporation of 3-D city modeling.

From these findings, the current allocation of 5,000 volunteers across various city locations, coupled with the potential addition of 7,000 more, showcases a compelling case for adopting an intensity-driven Poisson point process. The goal is to recruit 12,000 volunteers to maximize survival, and to recruit these volunteers at optimal locations. This approach, when matched with real-world data and population proportions, can guide and streamline recruiting efforts for better outcomes and more lives saved in OHCAs.

Yet, an undeniable aspect of this study remains the current model's limitation in accounting for NYC's 3-D architectural uniqueness. While the model provides substantial groundwork, the verticality of urban structures necessitates additional considerations. Enhancing the model to capture these nuances presents the next step in optimizing response to OHCAs in dense urban environments like NYC.

References

[Fir22] Fire Department, City of New York (FDNY). Mayor's management report 2022: Fdny, 2022. Available: https://www.nyc.gov/assets/operations/downloads/pdf/pmmr2022/fdny.pdf.

[Hen23] Shane Henderson. Course notes in orie 4130: Service system modeling, 2023. Unpublished course notes, Cornell University.

[Ope23] OpenAI. Chatgpt, 2023. Source for code and writing help.

[vdBHJL21] Pieter van den Berg, Shane G. Henderson, Caroline Jagtenberg, and Hemeng Li. Modeling emergency medical service volunteer response, April 11, 2021. Available at SSRN: https://ssrn.com/abstract=3825060 or http://dx.doi.org/10.2139/ssrn.3825060.

[Wik22] Wikipedia. New york city, 2022. Available: https://en.wikipedia.org/wiki/New_York_City.

Ope23 Fir22 Hen23 Wik22 [vdBHJL21]

APPENDIX

imports

In [2]:
```
!pip install --upgrade bokeh jinja2 python-dateutil packaging matplotlib tornado pillow numpy ty
!pip install haversine
!pip install statsmodels
!pip install bokeh
!pip install gurobipy



import numpy as np
!pip install scipy
import scipy as scs
import scipy.linalg as scl
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
import scipy
import pandas as pd
from shapely.geometry import Point
import geopandas as gpd
from geopandas import GeoDataFrame
# from google.colab import auth
# from pydrive.drive import GoogleDrive
# from pydrive.auth import GoogleAuth
# from oauth2client.client import GoogleCredentials
from markupsafe import Markup
from haversine import haversine
from matplotlib import cm
import matplotlib.colors
import math
from pyproj import Transformer
import copy
import itertools
import networkx as nx
from bokeh.plotting import figure, show
from bokeh.io import output_notebook
from bokeh.tile_providers import get_provider, Vendors
from bokeh.models import (GraphRenderer, Circle, MultiLine, StaticLayoutProvider,
                          HoverTool, TapTool, EdgesAndLinkedNodes,
                          NodesAndLinkedEdges, ColumnDataSource, LabelSet, NodesOnly)
from gurobipy import Model, GRB, quicksum
import numpy as np
import pandas as pd
!pip install rtree
import rtree
from scipy.spatial.distance import cdist
import numpy as np
import pandas as pd
from scipy.spatial.distance import cdist
```

```
Requirement already satisfied: bokeh in c:\users\julia\anaconda3\envs\engri_1101\lib\site-packag
es (2.4.3)
Requirement already satisfied: jinja2 in c:\users\julia\anaconda3\envs\engri_1101\lib\site-packa
ges (3.1.2)
Requirement already satisfied: python-dateutil in c:\users\julia\anaconda3\envs\engri_1101\lib\s
ite-packages (2.8.2)
Requirement already satisfied: packaging in c:\users\julia\anaconda3\envs\engri_1101\lib\site-pa
ckages (23.2)
Requirement already satisfied: matplotlib in c:\users\julia\anaconda3\envs\engri_1101\lib\site-p
ackages (3.5.3)
Requirement already satisfied: tornado in c:\users\julia\anaconda3\envs\engri_1101\lib\site-pack
ages (6.2)
Requirement already satisfied: pillow in c:\users\julia\anaconda3\envs\engri_1101\lib\site-packa
ges (9.5.0)
Requirement already satisfied: numpy in c:\users\julia\anaconda3\envs\engri_1101\lib\site-packag
es (1.21.6)
Requirement already satisfied: typing-extensions in c:\users\julia\anaconda3\envs\engri_1101\lib
\site-packages (4.7.1)
Requirement already satisfied: MarkupSafe in c:\users\julia\anaconda3\envs\engri_1101\lib\site-p
ackages (2.1.3)
Requirement already satisfied: PyYAML>=3.10 in c:\users\julia\anaconda3\envs\engri_1101\lib\site
-packages (from bokeh) (6.0)
Requirement already satisfied: six>=1.5 in c:\users\julia\anaconda3\envs\engri_1101\lib\site-pac
kages (from python-dateutil) (1.16.0)
Requirement already satisfied: kiwisolver>=1.0.1 in c:\users\julia\anaconda3\envs\engri_1101\lib
\site-packages (from matplotlib) (1.4.4)
Requirement already satisfied: cycler>=0.10 in c:\users\julia\anaconda3\envs\engri_1101\lib\site
-packages (from matplotlib) (0.11.0)
Requirement already satisfied: pyparsing>=2.2.1 in c:\users\julia\anaconda3\envs\engri_1101\lib
\site-packages (from matplotlib) (3.0.9)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\julia\anaconda3\envs\engri_1101\lib
\site-packages (from matplotlib) (4.38.0)
Requirement already satisfied: haversine in c:\users\julia\anaconda3\envs\engri_1101\lib\site-pa
ckages (2.8.0)
Requirement already satisfied: statsmodels in c:\users\julia\anaconda3\envs\engri_1101\lib\site-
packages (0.13.5)
Requirement already satisfied: scipy<1.8,>=1.3 in c:\users\julia\anaconda3\envs\engri_1101\lib\s
ite-packages (from statsmodels) (1.5.4)
Requirement already satisfied: patsy>=0.5.2 in c:\users\julia\anaconda3\envs\engri_1101\lib\site
-packages (from statsmodels) (0.5.3)
Requirement already satisfied: pandas>=0.25 in c:\users\julia\anaconda3\envs\engri_1101\lib\site
-packages (from statsmodels) (1.3.5)
Requirement already satisfied: packaging>=21.3 in c:\users\julia\anaconda3\envs\engri_1101\lib\s
ite-packages (from statsmodels) (23.2)
Requirement already satisfied: numpy>=1.17 in c:\users\julia\anaconda3\envs\engri_1101\lib\site-
packages (from statsmodels) (1.21.6)
Requirement already satisfied: pytz>=2017.3 in c:\users\julia\anaconda3\envs\engri_1101\lib\site
-packages (from pandas>=0.25->statsmodels) (2022.7)
Requirement already satisfied: python-dateutil>=2.7.3 in c:\users\julia\anaconda3\envs\engri_110
1\lib\site-packages (from pandas>=0.25->statsmodels) (2.8.2)
Requirement already satisfied: six in c:\users\julia\anaconda3\envs\engri_1101\lib\site-packages
(from patsy>=0.5.2->statsmodels) (1.16.0)
Requirement already satisfied: bokeh in c:\users\julia\anaconda3\envs\engri_1101\lib\site-packag
es (2.4.3)
Requirement already satisfied: packaging>=16.8 in c:\users\julia\anaconda3\envs\engri_1101\lib\s
ite-packages (from bokeh) (23.2)
Requirement already satisfied: typing-extensions>=3.10.0 in c:\users\julia\anaconda3\envs\engri_
1101\lib\site-packages (from bokeh) (4.7.1)
Requirement already satisfied: numpy>=1.11.3 in c:\users\julia\anaconda3\envs\engri_1101\lib\sit
e-packages (from bokeh) (1.21.6)
Requirement already satisfied: Jinja2>=2.9 in c:\users\julia\anaconda3\envs\engri_1101\lib\site-
packages (from bokeh) (3.1.2)
Requirement already satisfied: pillow>=7.1.0 in c:\users\julia\anaconda3\envs\engri_1101\lib\sit
e-packages (from bokeh) (9.5.0)
```

```
Requirement already satisfied: PyYAML>=3.10 in c:\users\julia\anaconda3\envs\engri_1101\lib\site
-packages (from bokeh) (6.0)
Requirement already satisfied: tornado>=5.1 in c:\users\julia\anaconda3\envs\engri_1101\lib\site
-packages (from bokeh) (6.2)
Requirement already satisfied: MarkupSafe>=2.0 in c:\users\julia\anaconda3\envs\engri_1101\lib\s
ite-packages (from Jinja2>=2.9->bokeh) (2.1.3)
Requirement already satisfied: gurobipy in c:\users\julia\anaconda3\envs\engri_1101\lib\site-pac
kages (10.0.3)
Requirement already satisfied: scipy in c:\users\julia\anaconda3\envs\engri_1101\lib\site-packag
es (1.5.4)
Requirement already satisfied: numpy>=1.14.5 in c:\users\julia\anaconda3\envs\engri_1101\lib\sit
e-packages (from scipy) (1.21.6)
Requirement already satisfied: rtree in c:\users\julia\anaconda3\envs\engri_1101\lib\site-packag
es (1.0.1)
Requirement already satisfied: typing-extensions>=3.7 in c:\users\julia\anaconda3\envs\engri_110
1\lib\site-packages (from rtree) (4.7.1)
```

uploading files

In [3]:

```
population_area = pd.read_csv('nyc_population_area.csv',low_memory=False, lineterminator='\n',in
population_area = population_area.reset_index()

nodes = pd.read_csv('nyc_nodes (1).csv',low_memory=False, lineterminator='\n',index_col = 0)
nodes = nodes.reset_index()

links = pd.read_csv('nyc_links (1).csv',low_memory=False, lineterminator='\n',index_col = 0)
links = links.reset_index()

population = pd.read_csv('nyc_population (1).csv',low_memory=False, lineterminator='\n',index_co
population = population.reset_index()
```

In [11]:

```
output_notebook()
def plotNetwork(nodes, links, dis_time, criteria, population, title='Plot of Graph', target=None
    """
    Plots a static map of a network = (nodes, links).
    :param nodes: pandas df with 'name', 'x', 'y' cols
                  'x' and 'y' treated as mercator coords
    :param links: pandas df with 'start' and 'end' cols
                  with entries matching nodes' names
    :param title: str of graph title
    :param target: list of node names
    :param on_map: boolean for map background
    """

    nodes = nodes
    links = links
    dfs = dis_time
    population = population
    speed = dfs['distance'].values/dfs['time'].values.tolist()

    if len(criteria)>0:
        groups = len(criteria)

        color = cm.get_cmap('Blues', groups-2)    # PiYG
        c_map = {}
        c_map[0] = '#ff0000'
        #c_map[1] = '#99c199'
        for i in range(color.N):
            rgba = color(i)
            # rgb2hex accepts rgb or rgba
            c_map[i+1] = matplotlib.colors.rgb2hex(rgba)
        c_map[i+2] = '#FFFF00'
```

```
    e_clr = []
    for i in speed:
        for k in range(groups):
            if i>=criteria[k][0] and i<=criteria[k][1]:
                e_clr.append(c_map[k])
                break

else:
    e_clr = []
    for i in speed:
            e_clr.append('#A0A0A0')


# extract data
node_ids = nodes.name.values.tolist()
start = links.start.values.tolist()
end = links.end.values.tolist()
x = nodes.x.values.tolist()
y = nodes.y.values.tolist()


# get plot boundaries
min_x, max_x = min(nodes.x)+2000, max(nodes.x)-2000
min_y, max_y = min(nodes.y)+2000, max(nodes.y)-2000

plot = figure(x_range=(min_x, max_x), y_range=(min_y, max_y),
              x_axis_type="mercator", y_axis_type="mercator",
              title=title,
              width=600, height=470,
              toolbar_location=None, tools=[]
             )

graph = GraphRenderer()

if on_map == True:
    # add map tile
    plot.add_tile(get_provider(Vendors.CARTODBPOSITRON_RETINA))

# define nodes
graph.node_renderer.data_source.add(node_ids, 'index')
graph.node_renderer.glyph = Circle(line_color='green', line_alpha=0,
                                   fill_color='green', size=3.5,
                                   fill_alpha=0
                                  )


# define edges
graph.edge_renderer.data_source.data = dict(start=list(start),
                                            end=list(end),e_clr=e_clr
                                           )
graph.edge_renderer.glyph = MultiLine(line_color = 'e_clr',
                                      line_alpha=1, line_width=.6
                                     )

# set node locations
graph_layout = dict(zip(node_ids, zip(x, y)))
graph.layout_provider = StaticLayoutProvider(graph_layout=graph_layout)

plot.renderers.append(graph)


# add POIS
```

```
    if len(population)>0:
        color = cm.get_cmap('Reds', 7)
        for i in range(7):
            target = []
            for index,row in population.iterrows():
                if row['pop']>=(i*2500) and row['pop']<((i+1)*2500):
                    target.append(row['name'])

            populationp = nodes.loc[nodes['name'].isin(target)]
            source = ColumnDataSource(populationp)
            poi = Circle(x="x", y="y", size=3.5, line_color='blue',
                         fill_color=matplotlib.colors.rgb2hex(color(i)), line_width=0.1
                        )

            plot.add_glyph(source, poi)

    show(plot)
```


Loading BokehJS ...

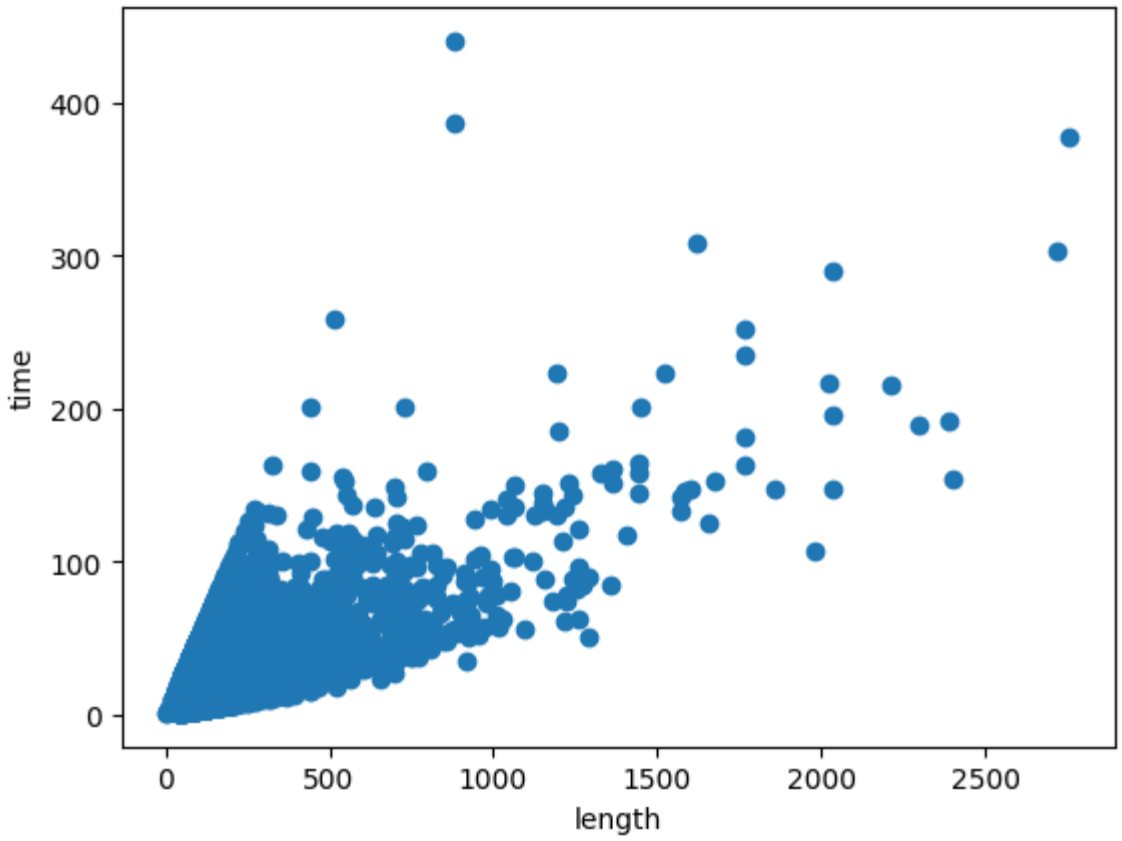
assume volunteer alerted 3 minutes after OHCA

assume 30% probability acceptance

In [12]:
```
response_delay = 3 #3 mins after OHCA for volunteer to be alerted
probability_accept = 0.3
```

In [13]:
```
# #plotting
dfs = pd.DataFrame(columns = ["distance", "time"])

for index,row in links.iterrows():
    start_point = row['start']
    end_point = row['end']
    start_point_coord_row = nodes[(nodes.name==start_point)].index[0]
    end_point_coord_row = nodes[(nodes.name==end_point)].index[0]
    start_point_coord = nodes.loc[start_point_coord_row,['lat','lon']].tolist()
    end_point_coord = nodes.loc[end_point_coord_row,['lat','lon']].tolist()
    dfs.loc[index] = [haversine(start_point_coord, end_point_coord)*1000,row['time']]


#correcting travel times
dfs_new = copy.deepcopy(dfs)    #new data with adjusted time
for index,row in dfs_new.iterrows():
    if row['distance']/row['time']<2:
        row['time'] = row['distance']/2
    else:
        if row['distance']/row['time']>30:
            row['time'] = row['distance']/30
        start = links.loc[index]['start']
        end = links.loc[index]['end']
        start_loc = nodes[nodes['name']==start][['lat','lon']].values.tolist()[0]
        end_loc = nodes[nodes['name']==end][['lat','lon']].values.tolist()[0]

        if (0<(start_loc[0]-40.66)/(start_loc[1]+74.05)<2 or 0<(end_loc[0]-40.66)/(end_loc[1]+74
            row['time'] = row['distance']/20

plt.xlabel("length")
plt.ylabel("time")
```

```
plt.scatter(dfs_new['distance'], dfs_new['time'])
plt.show()
```

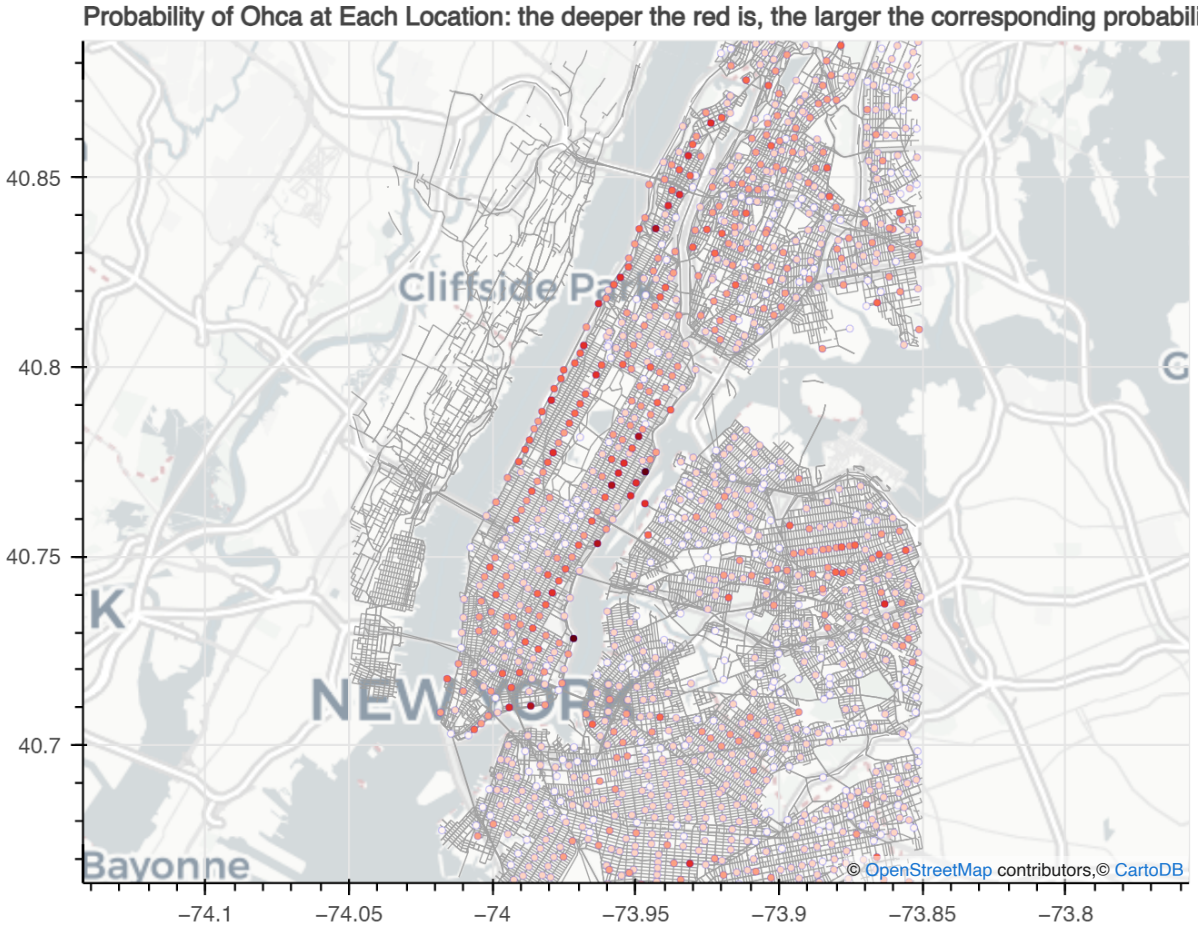


In [14]:
```
#ambulance speeds
ambulance_dfl = copy.deepcopy(links)    #travel time for ambulance
ambulance_dfl['time'] = 0.9*dfs_new['time']
```

In [16]:
```
#probability from population
total_population = sum(population['pop'].tolist())
call_prob = []
for index,row in population.iterrows():
    call_prob.append(row['pop']/total_population)
population_with_prob = copy.deepcopy(population) #a new dataframe with a new column showing the
population_with_prob['prob'] = call_prob
```

probability of OHCA at each location

In [17]:
```
plotNetwork(nodes, links,dfs, [], population, title="Probability of Ohca at Each Location: the d
```



In [18]:
```
population = population_area
population['probability_ohca']= population_with_prob['prob']
population['people_per_sq_m'] = population['pop']/population['area (m^2)\r']
population['area km^2'] = population['area (m^2)\r']*0.000001
population['people_per_sq_km'] = population['pop']/population['area km^2']
```

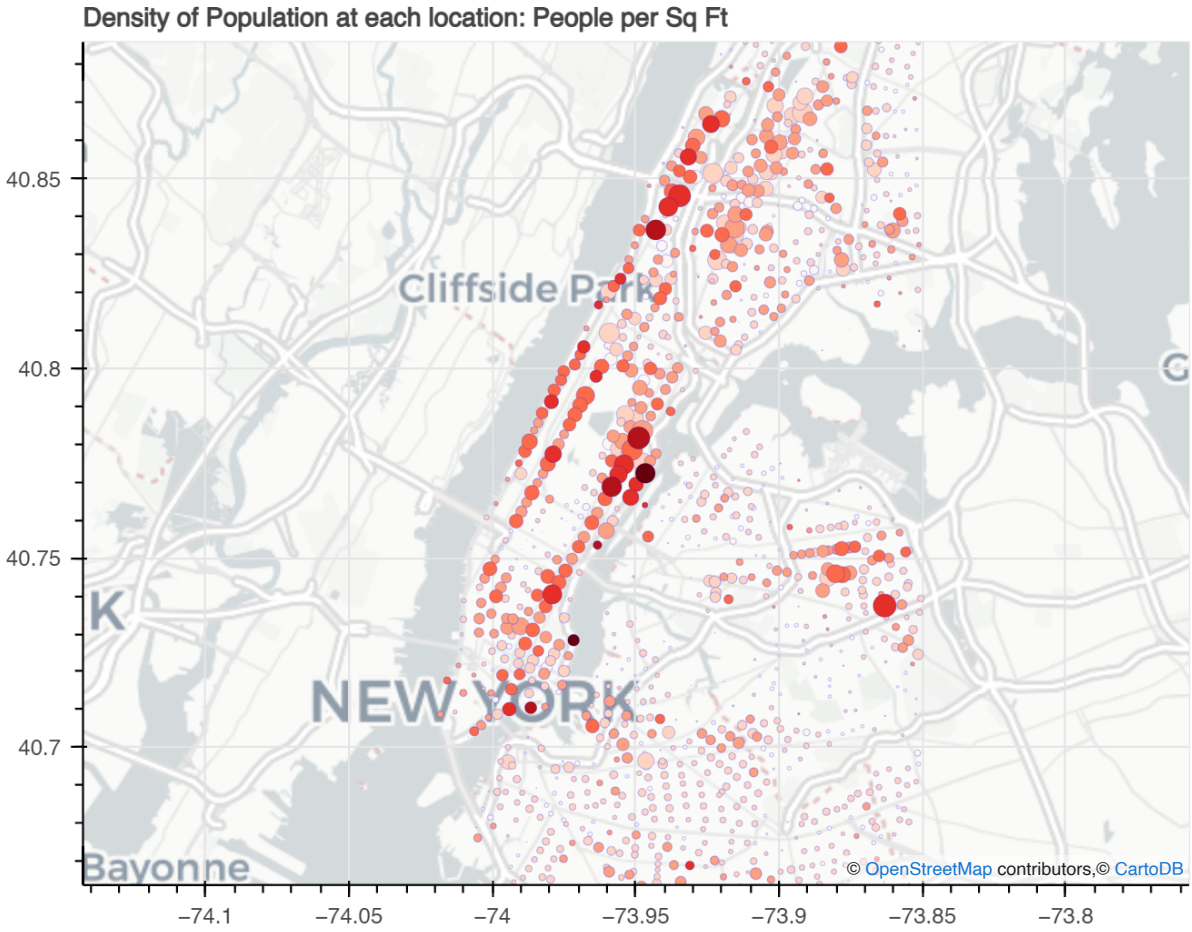
In [19]:
```
output_notebook()
def plotNetwork(nodes, links, dis_time, criteria, population, title='Plot of Graph', target=None
    """
    Plots a static map of a network = (nodes, links).
    :param nodes: pandas df with 'name', 'x', 'y' cols
                  'x' and 'y' treated as mercator coords
    :param links: pandas df with 'start' and 'end' cols
                  with entries matching nodes' names
    :param title: str of graph title
    :param target: list of node names
    :param on_map: boolean for map background
    """
    dfn = nodes
    dfl = links
    dfs = dis_time
    dfp = population
    speed = dfs['distance'].values/dfs['time'].values.tolist()
    if len(criteria)>0:
        groups = len(criteria)

        color = cm.get_cmap('Blues', groups-2)    # PiYG
        c_map = {}
        c_map[0] = '#ff0000'
        #c_map[1] = '#99c199'
        for i in range(color.N):
            rgba = color(i)
            # rgb2hex accepts rgb or rgba
```

```
            c_map[i+1] = matplotlib.colors.rgb2hex(rgba)
        c_map[i+2] = '#FFFF00'
    # extract data
    node_ids = dfn.name.values.tolist()
    start = dfl.start.values.tolist()
    end = dfl.end.values.tolist()
    x = dfn.x.values.tolist()
    y = dfn.y.values.tolist()
    # get plot boundaries
    min_x, max_x = min(dfn.x)+2000, max(dfn.x)-2000
    min_y, max_y = min(dfn.y)+2000, max(dfn.y)-2000

    plot = figure(x_range=(min_x, max_x), y_range=(min_y, max_y),
                  x_axis_type="mercator", y_axis_type="mercator",
                  title=title,
                  width=600, height=470,
                  toolbar_location=None, tools=[]
                 )
    graph = GraphRenderer()
    if on_map == True:
        # add map tile
        plot.add_tile(get_provider(Vendors.CARTODBPOSITRON_RETINA))
    # define nodes
    graph.node_renderer.data_source.add(node_ids, 'index')
    graph.node_renderer.glyph = Circle(line_color='green', line_alpha=0,
                                       fill_color='green', size=3.5,
                                       fill_alpha=0
                                      )
    # set node locations
    graph_layout = dict(zip(node_ids, zip(x, y)))
    graph.layout_provider = StaticLayoutProvider(graph_layout=graph_layout)
    plot.renderers.append(graph)
    # add POIS
    if len(population)>0:
        color = cm.get_cmap('Reds', 7)
        for i in range(7):
            target = []
            for index,row in dfp.iterrows():
                if row['pop']>=(i*2500) and row['pop']<((i+1)*2500):
                    target.append(row['name'])
            dfpp = dfn.loc[dfn['name'].isin(target)]
            size = []
            for index,row in dfpp.iterrows():
                size_value = dfp[dfp['name'] == row['name']]['people_per_sq_km'].values[0] / 700
                size.append(size_value)
            dfpp['size'] = size
            source = ColumnDataSource(dfpp)
            poi = Circle(x="x", y="y", size='size', line_color='blue',
                         fill_color=matplotlib.colors.rgb2hex(color(i)), line_width=0.1
                        )
            plot.add_glyph(source, poi)
    show(plot)
```

Loading BokehJS ...

In [20]:
```
plotNetwork(nodes, links,dfs, population['area km^2'], population, title="Density of Population
```

```
C:\Users\julia\anaconda3\envs\engri_1101\lib\site-packages\ipykernel_launcher.py:73: SettingWith
CopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/in
dexing.html#returning-a-view-versus-a-copy
```



Plotting Volunteer Locations

In [21]:
```
result = pd.merge(nodes, population, on='name', how='inner')
result['volunteer_scale'] = result['people_per_sq_km']/sum(result['people_per_sq_km'])
result
```

Out[21]:

	name	lon	lat	x	y	pop	area (m^2)\r	probability_ohca	people_pe
0	42467159	-73.891217	40.672581	-8.225533e+06	4.964167e+06	3094	166136	0.000631	0
1	42467069	-73.931406	40.661331	-8.230006e+06	4.962516e+06	2863	182479	0.000584	0
2	42466853	-73.937942	40.726239	-8.230734e+06	4.972045e+06	2228	188108	0.000454	0
3	1413215971	-73.958524	40.672663	-8.233025e+06	4.964179e+06	3597	119678	0.000733	0
4	42466495	-73.986996	40.683765	-8.236195e+06	4.965808e+06	4495	167532	0.000917	0
...	...	...	...	...	...	...	...	...	
1158	42467478	-73.942696	40.664055	-8.231263e+06	4.962915e+06	4520	142464	0.000922	0
1159	42467474	-73.942528	40.665838	-8.231245e+06	4.963177e+06	4085	186338	0.000833	0
1160	42467470	-73.942361	40.667622	-8.231226e+06	4.963439e+06	3139	180682	0.000640	0
1161	42467460	-73.942030	40.671185	-8.231189e+06	4.963962e+06	4376	180739	0.000892	0
1162	42467300	-73.949930	40.654150	-8.232069e+06	4.961462e+06	5281	148600	0.001077	0

1163 rows × 12 columns

Probability OHCA is p_i.

n = # of volunteers

λ_i is expected number of volunteers at the ith dot. $\lambda_i = n \propto p_i$

we want E[# of volunteers within a radius s(t-180)

s = speed of volunteers in m/s = assume 6 km/h

divide NYC into 1163 "nodes" each with a corresponding area. probability of OHCA is proportional to the population of the area. volunteer density in the area is found by getting the number of people per sq km and scaling this with respect to the number of volunteers.

we model the OHCA demand rate p_i each location as being proportional to the population in that location

we use population area csv to get the population density per area. this gives us people per sq km in the areas we are studying.

estimate that 30% of volunteers accept the alert.

we do not have exact locations of volunteers, but we can assume that volunteer distribution in each area is proportional to population. and volunteer density is proportional to population density.

assume that volunteers walk at a constant speed of 6 km/h = 1.66667 m/s.

We assume a constant response delay of 3 minutes.

Assume only dispatch volunteers within 1km of the incident. The volunteer response time is then between 3 and 13 minutes.

From last case, we assume NYC is following deployment that we said last time. Assume constant 4 minutes after the call the ambulance is deployed (pre trip delay). 90% of calls are reached in 9 minutes.

time interval 7pm-midnight on weekdays.

We assume there are 177/2 calls per hour, and these are distributed according to population at a poisson point process.

We assume ambulances are sitting at the nodes we said in compliance table last time.

We distribute ambulances across the city according to the optimal solution of MEXCLP. We are assuming 30 minute hospital offload delay so we have 139 ambulances. They have a utilization rate of 0.74 (probability busy)

In [22]:
```
def generate_OHCA():
    nodes = result['name']
    probability = result['probability_ohca']
    node_OHCA= np.random.choice(nodes,p=probability)
    return result[result['name'] == node_OHCA]
```

In [23]:
```
# Written by Chat GPT

def generate_volunteers_with_coordinates(N):
    # Calculate the maximum population density among all nodes
    max_density = result['people_per_sq_km'].max()

    # Initialize a dictionary to store the number of volunteers for each node
    node_volunteer_count = {node_name: 0 for node_name in result['name']}

    # Set the desired total number of volunteers
    total_volunteers = N

    # Initialize lists to store volunteer coordinates
    volunteer_coordinates = []

    # Loop to generate volunteers
    while sum(node_volunteer_count.values()) < total_volunteers:
        node = np.random.choice(result['name'], p=result['probability_ohca'])
        lon = float(result[result['name'] == node]['lon'])
        lat = float(result[result['name'] == node]['lat'])

        # Assuming 'node' contains the name of the closest node
        closest_node_index = result[result['name'] == node].index[0]

        # Get the closest node
        closest_node = result.iloc[closest_node_index]

        # Calculate the probability of selecting this location based on population density
        probability = closest_node['people_per_sq_km'] / max_density

        # Generate a random number between 0 and 1
        random_number = np.random.uniform(0, 1)

        # Accept the location if random_number < probability
        if random_number < probability:
            # Update the count of volunteers for the selected node
            node_volunteer_count[closest_node['name']] += 1

            # Generate coordinates within a circle around the volunteer location
            center_lat = closest_node['lat']
            center_lon = closest_node['lon']
```

```
            radius_meters = np.sqrt(closest_node['area (m^2)\r'] / np.pi)
            num_points = 1  # One volunteer per location

            # Earth's radius in meters
            earth_radius = 6371000.0  # Approximate value for Earth's radius

            # Generate random radii and angles for each point
            radii = np.sqrt(np.random.uniform(0, 1, num_points)) * radius_meters
            angles = np.random.uniform(0, 2 * np.pi, num_points)

            # Calculate latitude and longitude offsets for each point
            lat_offsets = radii * np.sin(angles) / earth_radius
            lon_offsets = radii * np.cos(angles) / (earth_radius * np.cos(np.radians(center_lat)

            # Calculate the final latitude and longitude coordinates
            latitudes = center_lat + np.degrees(lat_offsets)
            longitudes = center_lon + np.degrees(lon_offsets)

            # Append the coordinates to the volunteer_coordinates list
            volunteer_coordinates.append({'longitude': longitudes[0], 'latitude': latitudes[0]})

    # Create a DataFrame with the node name and the number of volunteers
    volunteer_count_df = pd.DataFrame({'node_name': list(node_volunteer_count.keys()),
                                       'number_of_volunteers': list(node_volunteer_count.values(

    # Create a DataFrame with the volunteer coordinates
    volunteer_coordinate_df = pd.DataFrame(volunteer_coordinates)

    return volunteer_count_df, volunteer_coordinate_df
```

For each OHCA, available volunteers around OHCA is possion with mean

In [24]:
```
volunteer_count_df, volunteer_coordinate_df = generate_volunteers_with_coordinates(5000)
```

In [25]:
```
import numpy as np
def merc_from_arrays(lats, lons):
    r_major = 6378137.000
    x = r_major * np.radians(lons)
    scale = x/lons
    y = 180.0/np.pi * np.log(np.tan(np.pi/4.0 + lats * (np.pi/180.0)/2.0)) * scale
    return (x, y)

volunteer_coordinate_df['x'],volunteer_coordinate_df['y'] = merc_from_arrays(volunteer_coordinat
```

In [26]:
```
output_notebook()
def plotNetwork(nodes, links, dis_time, criteria, population, volunteer_coordinate_df, title='Pl
    """
    Plots a static map of a network = (nodes, links).
    :param nodes: pandas df with 'name', 'x', 'y' cols
                  'x' and 'y' treated as mercator coords
    :param links: pandas df with 'start' and 'end' cols
                  with entries matching nodes' names
    :param title: str of graph title
    :param target: list of node names
    :param on_map: boolean for map background
    """


    nodes = nodes
    links = links
    dfs = dis_time
    population = population
    speed = dfs['distance'].values/dfs['time'].values.tolist()
    volunteers = volunteer_coordinate_df
```

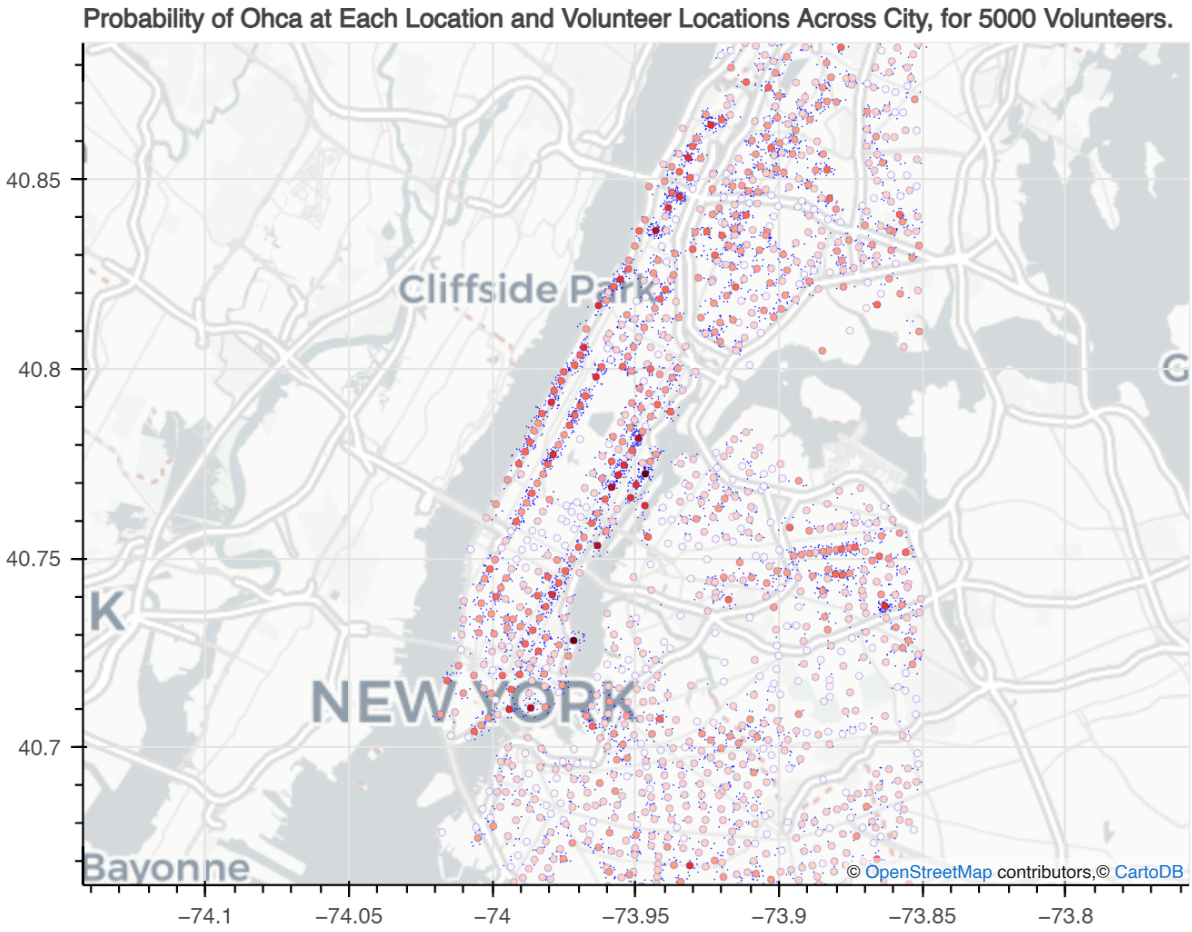
```
node_ids_v = volunteers.index.values.tolist()
x_v = volunteers.x.values.tolist()
y_v = volunteers.y.values.tolist()

if len(criteria)>0:
    groups = len(criteria)

    color = cm.get_cmap('Blues', groups-2)    # PiYG
    c_map = {}
    c_map[0] = '#ff0000'
    #c_map[1] = '#99c199'
    for i in range(color.N):
        rgba = color(i)
        # rgb2hex accepts rgb or rgba
        c_map[i+1] = matplotlib.colors.rgb2hex(rgba)
    c_map[i+2] = '#FFFF00'


    e_clr = []
    for i in speed:
        for k in range(groups):
            if i>=criteria[k][0] and i<=criteria[k][1]:
                e_clr.append(c_map[k])
                break

else:
    e_clr = []
    for i in speed:
            e_clr.append('#A0A0A0')


# extract data
node_ids = nodes.name.values.tolist()
start = links.start.values.tolist()
end = links.end.values.tolist()
x = nodes.x.values.tolist()
y = nodes.y.values.tolist()


# get plot boundaries
min_x, max_x = min(nodes.x)+2000, max(nodes.x)-2000
min_y, max_y = min(nodes.y)+2000, max(nodes.y)-2000

plot = figure(x_range=(min_x, max_x), y_range=(min_y, max_y),
              x_axis_type="mercator", y_axis_type="mercator",
              title=title,
              width=600, height=470,
              toolbar_location=None, tools=[]
             )

graph = GraphRenderer()

if on_map == True:
    # add map tile
    plot.add_tile(get_provider(Vendors.CARTODBPOSITRON_RETINA))

# define nodes
graph.node_renderer.data_source.add(node_ids, 'index')
graph.node_renderer.data_source.add(node_ids_v, 'index_v')
graph.node_renderer.glyph = Circle(line_color='green', line_alpha=0,
                                   fill_color='green', size=3.5,
                                   fill_alpha=0
                                  )
```

```
    # define edges
    # graph.edge_renderer.data_source.data = dict(start=list(start),
    #                                             end=list(end),e_clr=e_clr
    #                                            )
    # graph.edge_renderer.glyph = MultiLine(line_color = 'e_clr',
    #                                       line_alpha=1, line_width=.6
    #                                      )

    # set node locations
    graph_layout = dict(zip(node_ids, zip(x, y)))
    graph.layout_provider = StaticLayoutProvider(graph_layout=graph_layout)

    plot.renderers.append(graph)

    source_v = ColumnDataSource(volunteers)
    vols = Circle(x="x", y="y", size=.5, line_color='blue',
                        fill_color='blue', line_width=0.1
                       )
    plot.add_glyph(source_v, vols)


    # add POIS
    if len(population)>0:
        color = cm.get_cmap('Reds', 7)
        for i in range(7):
            target = []
            for index,row in population.iterrows():
                if row['pop']>=(i*2500) and row['pop']<((i+1)*2500):
                    target.append(row['name'])

            populationp = nodes.loc[nodes['name'].isin(target)]
            source = ColumnDataSource(populationp)
            poi = Circle(x="x", y="y", size=3.5, line_color='blue',
                        fill_color=matplotlib.colors.rgb2hex(color(i)), line_width=0.1
                       )

            plot.add_glyph(source, poi)

    show(plot)
```

Loading BokehJS ...

In [27]:
```
plotNetwork(nodes, links,dfs, [], population, volunteer_coordinate_df, title="Probability of Ohc
```

```
BokehUserWarning: ColumnDataSource's columns must be of the same length. Current lengths: ('inde
x', 20056), ('index_v', 5000)
```



error in volunteer generation where some volunteers are not on land. Simplifying assumption

In [28]:
```
def survival_rate(t):
    return (1+np.e**(0.679+0.262*t))**(-1)
```

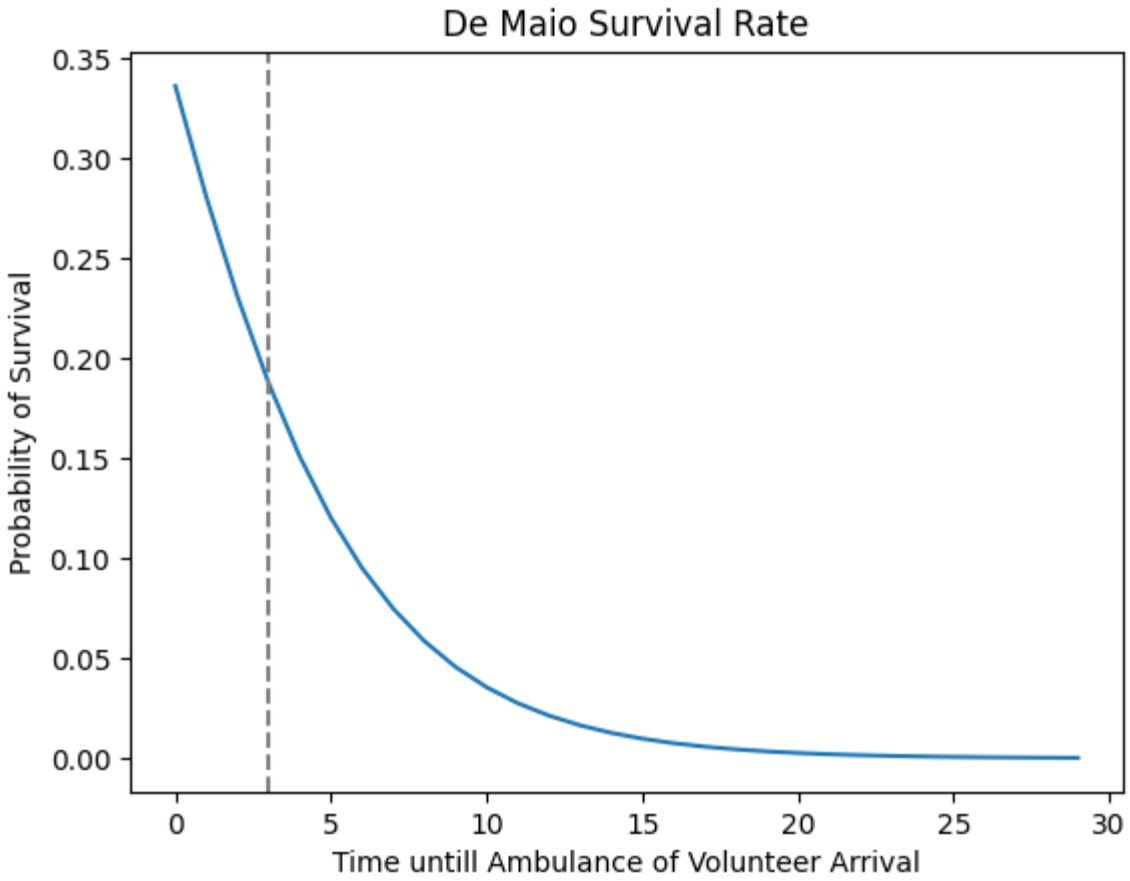
start with straight lines. Assume ambulances have response time distribution of triangular with min 3, max 12, and mode 9

Todo: generate OHCA generate volunteers probability accept find the closest ambulance and get the time it will arrive. use compliance table. or could just use 9 minute assumption -- normally distributed business with utilization rate as shown. get the walking time if the people from the map?? could just do this assuming straight lines

In [30]:
```
survival=[]
for i in np.arange(0,30,1):
    survival.append(survival_rate(i))

plt.plot(np.arange(0,30,1),survival)
plt.axvline(3, color='grey', linestyle='--')
plt.xlabel('Time untill Ambulance of Volunteer Arrival')
```

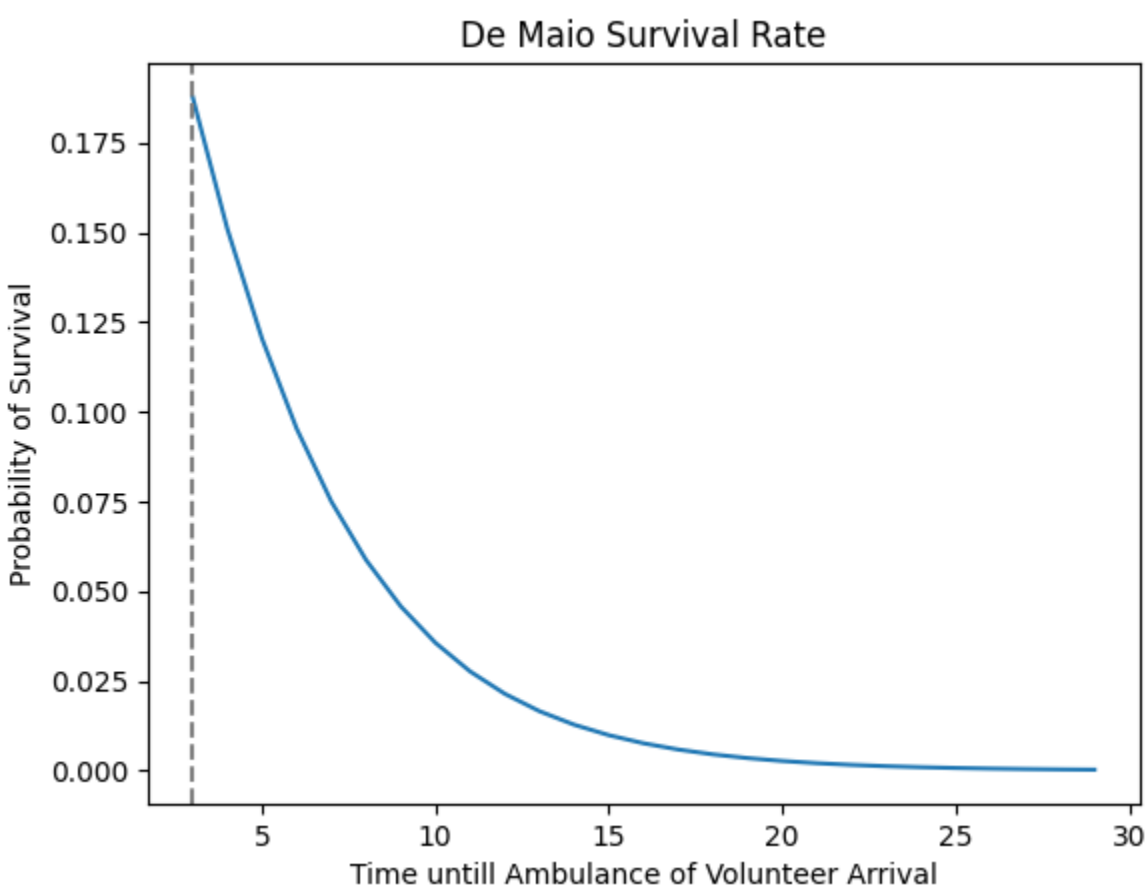
```
plt.ylabel('Probability of Survival')
plt.title('De Maio Survival Rate')
```

Out[30]:
```
Text(0.5, 1.0, 'De Maio Survival Rate')
```



In [31]:
```
survival=[]
for i in np.arange(3,30,1):
    survival.append(survival_rate(i))

plt.plot(np.arange(3,30,1),survival)
plt.axvline(3, color='grey', linestyle='--')
plt.xlabel('Time untill Ambulance of Volunteer Arrival')
plt.ylabel('Probability of Survival')
plt.title('De Maio Survival Rate')
```

Out[31]:
```
Text(0.5, 1.0, 'De Maio Survival Rate')
```



Volunteer

Volunteer density is is proportional to the population density (people per square kilometer) in that location.

We will use the De Maio survival function to define survival rate, which uses one variable t being the time between call arrival and either ambulance arrival or volunteer arrival, measured in minutes.

This function is:

$$s(t) = (1 + e^{0.697+0.262t})^{-1}$$

a volunteer is available and accepts the offer with probability 0.3

Avalable volunteers are distributed throughout NYC according to a spacial Poisson point process

Assuption that available volunteers does not depend on number of calls since volunteers are only alerted about once per year and OHCAs are not very common events.

When the number of volunteers n is large, a Poisson point process model is appropriate.

If we have N volunteers across the city, we have 0.3N available volunteers.

This will converge to a poisson process of mean λ where

$\lim_{n\to\infty} \sum_{i=1}^{n} \alpha_i(n) v_i(B) = \lambda(B)$

where $\alpha_i(n)$ is the probability that volunteer i is available when there are n registered volunteers, and v_i is the location distribution of volunteer i in the city.

We then set $\lambda = n\alpha v$, with α=0.3 for all volunteers across the city.

The location probability distribution will be the same probability distribution as OHCAs, proportional to population in each location (node here).

Then, we need to find the distribution of response times t of the closest volunteer.

t will be the time until the first volunteer or ambulance arrives. So, $t = min(t_{vol}, t_{amb})$

We know $t_{amb} = triangular(3, 9, 12)$

So, to get t_{vol}, we have $t_{vol} = responsedelay + walkingtime$

We know $responsedelay = 3$ min.

So, to get t_{walk}, we can define the region around the OHCA i as region j. Lets define this region to be $R(i, j)$.

Then, we know that this is also a poisson point pricess with mean $\lambda(R(i, j)) = n\alpha v(R(i, j))$

From study of poisson process, we know that $P(T_{walk} \leq t) = 1 - e^{-\lambda R(i,j)}$

this will give us the probability that the response time of the closest volunteer is less than t.

We then can set λ_l to be the constant available volunteer density in the area.

So, $P(T_{walk} \leq t) = 1 - e^{-\lambda_l \pi d_t^2}$

where $d_t = s(t - T_{walk})$ and s is volunteer walking speed in km/minute. We assume speed is 6km/hr or 0.1km/min.

Since we know that volunteer response delay is 3 minutes, We then get, for the probability that there is

$$P(T_{walk} \leq t) = 1 - e^{-\lambda_l \pi s^2 (t-3)^2}$$

And here, λ_l = available volunteer density in the area, in volunteers per km.

so, λ_l = (Total Volunteers in City) / (Total population of City) * (Population Density in Area) and Population Density in area = (Total Population in area) / (Total Area)

so, we define p_l to be population in area l, a_l to be the area in km^2 of area l, N total volunteers, and P to be total population in the city, and A to be the area of the city in km^2.

So, $\lambda_l = .3Np_l/Pa_l$

$$P(T_{walk_l} \leq t) = 1 - e^{-\frac{.3Np_l}{Pa_l} \pi s^2 (t-3)^2}$$

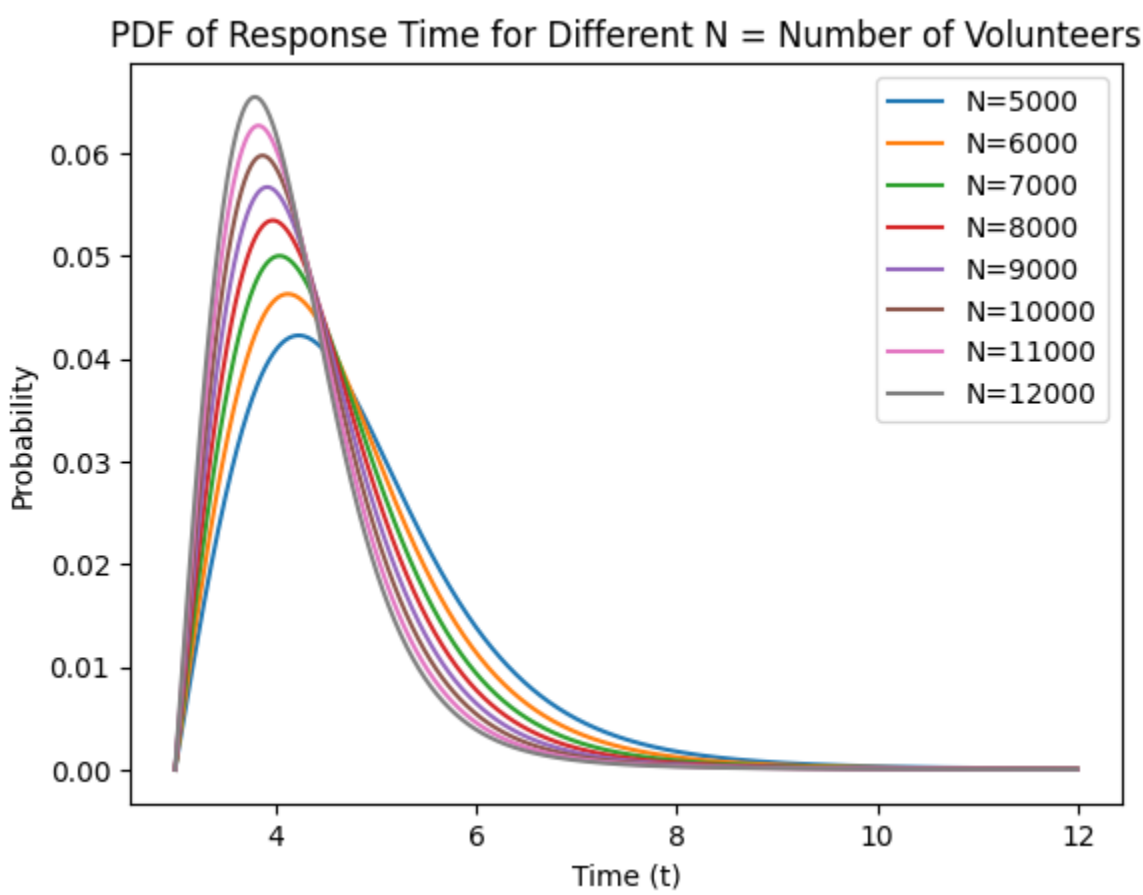
Then, to get the CDF of all volunteer response times, We have a probability that a call arises in a given region, P_l, and $\sum_l P_l = 1$

So, $P(T_v \leq t) = \int_l P_l 1 - e^{-\frac{.3Np_l}{Pa_l} \pi s^2 (t-3)^2}$

The PDF of this function is $f(t) = \int_l P_l \frac{2\pi .3Np_1 s^2 \cdot (t-3)\, \mathrm{e}^{-\frac{\pi .3Np_1 s^2 \cdot (t-3)^2}{Pa_1}}}{Pa_1}$

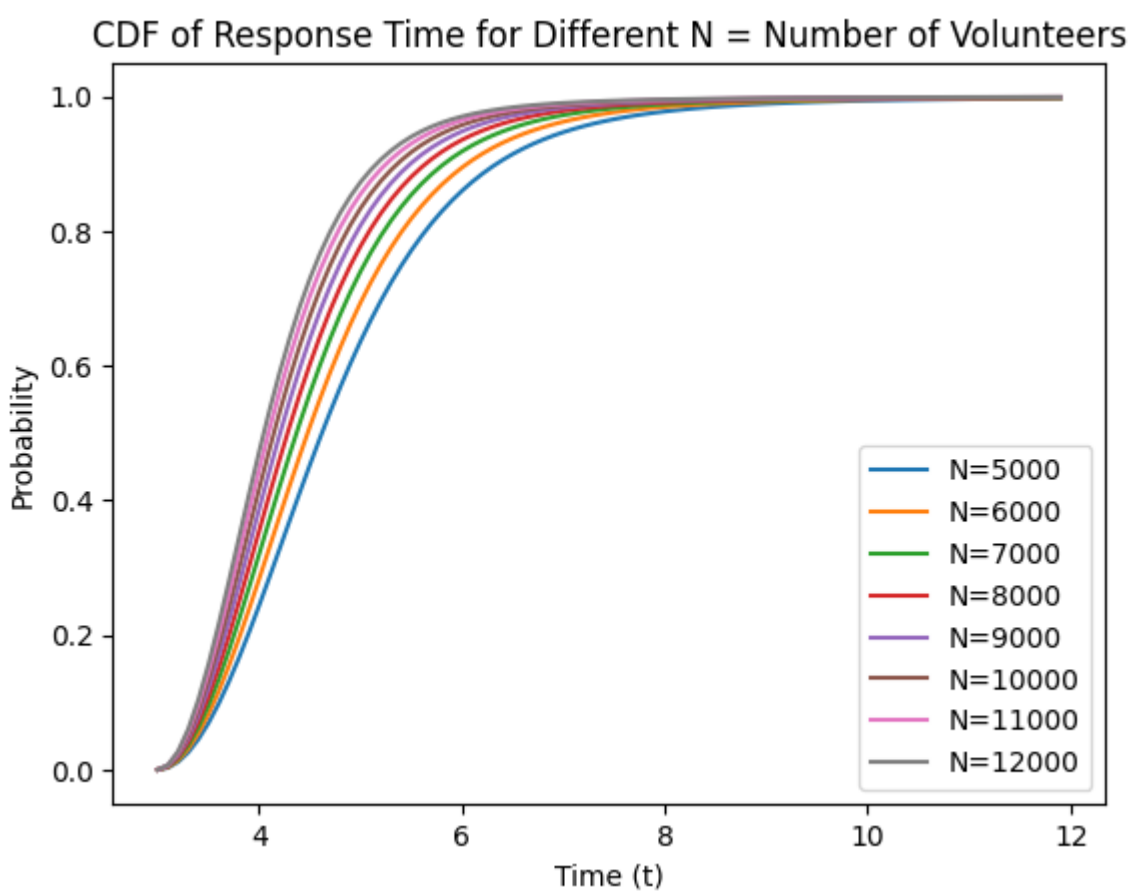
In [32]:
```
import numpy as np

# Values of N ranging from 1000 to 12001
N_values = np.arange(5000, 12001, 1000)

# Extract necessary columns from the DataFrame once
pop_sum = result['pop'].sum()
prob_ohca_values = result['probability_ohca'].values
people_per_sq_km_values = result['people_per_sq_km'].values

# Initialize an empty list to store the functions for different N values
functions = []
mean_times = []

# Pre-generate the time array
time_array = np.arange(3, 12, 0.01)

# Iterate over each N value
for N in N_values:
    function_values = []

    for t in time_array:
        exp_values = np.exp(-1 * N *.3* people_per_sq_km_values * np.pi * (0.1 ** 2) * (t - 3) *
        prob_call_values = 2*np.pi*N*.3*people_per_sq_km_values*(0.1 ** 2) * (t - 3) / pop_sum

        total_prob = np.sum(prob_ohca_values * prob_call_values * exp_values)/10
        function_values.append(total_prob)

    functions.append(function_values)

# After the loop, normalize and calculate mean_times
normalized_functions = [f/np.sum(f) for f in functions]
mean_times = [np.dot(nf, time_array) for nf in normalized_functions]
```

In [33]:
```
# Plot the functions for different N values
for i, N in enumerate(N_values):
    plt.plot(np.arange(3, 12, 0.01), functions[i], label=f'N={N}')

plt.xlabel('Time (t)')
plt.ylabel('Probability')
plt.title('PDF of Response Time for Different N = Number of Volunteers')
plt.legend()
plt.show()
```



In [62]:

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

# Values of N ranging from 5000 to 12000
N_values = np.arange(5000, 12001, 1000)

# Extract necessary columns from the DataFrame once
pop_sum = result['pop'].sum()
prob_ohca_values = result['probability_ohca'].values
people_per_sq_km_values = result['people_per_sq_km'].values

# Initialize an empty list to store the functions for different N values
functions = []

time_values = np.arange(3, 12, 0.1)

# Iterate over each N value
for N in N_values:
    exp_values = -1 * .3 * N * people_per_sq_km_values * np.pi * (0.1 ** 2) * np.outer((time_val
    prob_call_values = 1 - np.exp(exp_values)

    total_probs = np.dot(prob_call_values, prob_ohca_values)

    functions.append(
        total_probs)

# Plot the functions for different N values
for i, N in enumerate(N_values):
    plt.plot(time_values, functions[i], label=f'N={N}')

plt.xlabel('Time (t)')
plt.ylabel('Probability')
plt.title('CDF of Response Time for Different N = Number of Volunteers')
plt.legend()
plt.show()
```

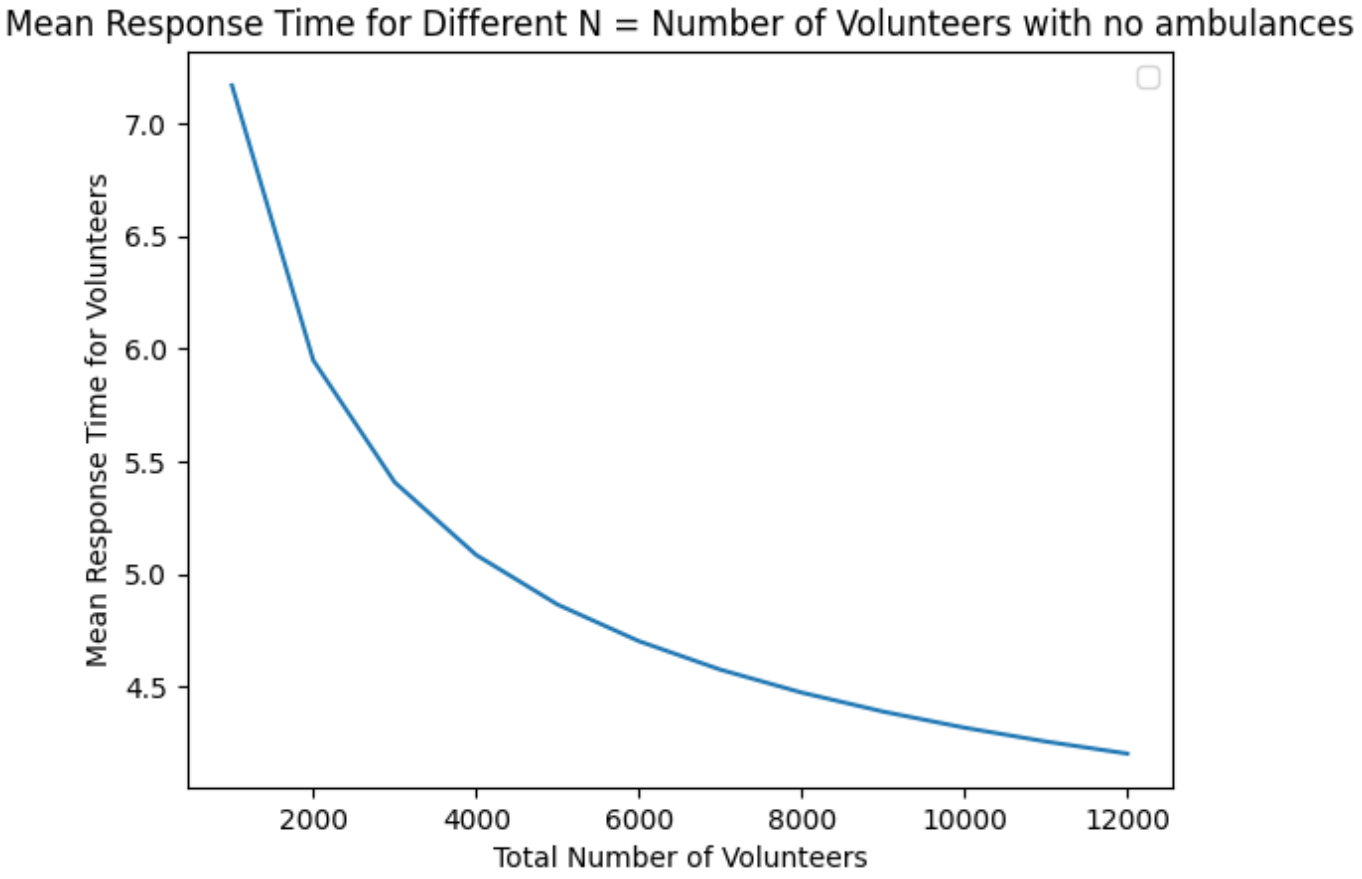


In [92]:

```python
import numpy as np

# Values of N ranging from 1000 to 12001
N_values = np.arange(1000, 12001, 1000)

# Extract necessary columns from the DataFrame once
pop_sum = result['pop'].sum()
prob_ohca_values = result['probability_ohca'].values
people_per_sq_km_values = result['people_per_sq_km'].values

# Initialize an empty list to store the functions for different N values
functions = []
mean_times = []

# Pre-generate the time array
time_array = np.arange(3, 100, 0.01)

# Iterate over each N value
for N in N_values:
    function_values = []

    for t in time_array:
        exp_values = np.exp(-1 * N *.3* people_per_sq_km_values * np.pi * (0.1 ** 2) * (t - 3) *
        prob_call_values = 2*np.pi*N*.3*people_per_sq_km_values*(0.1 ** 2) * (t - 3) / pop_sum

        total_prob = np.sum(prob_ohca_values * prob_call_values * exp_values)/10
        function_values.append(total_prob)

    functions.append(function_values)

# After the loop, normalize and calculate mean_times
normalized_functions = [f/np.sum(f) for f in functions]
mean_times = [np.dot(nf, time_array) for nf in normalized_functions]
```

In [93]:
```
plt.plot(np.arange(1000, 12001, 1000), mean_times)
plt.xlabel('Total Number of Volunteers')
plt.ylabel('Mean Response Time for Volunteers')
plt.title('Mean Response Time for Different N = Number of Volunteers with no ambulances')
plt.legend()
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an unde
rscore are ignored when legend() is called with no argument.
```

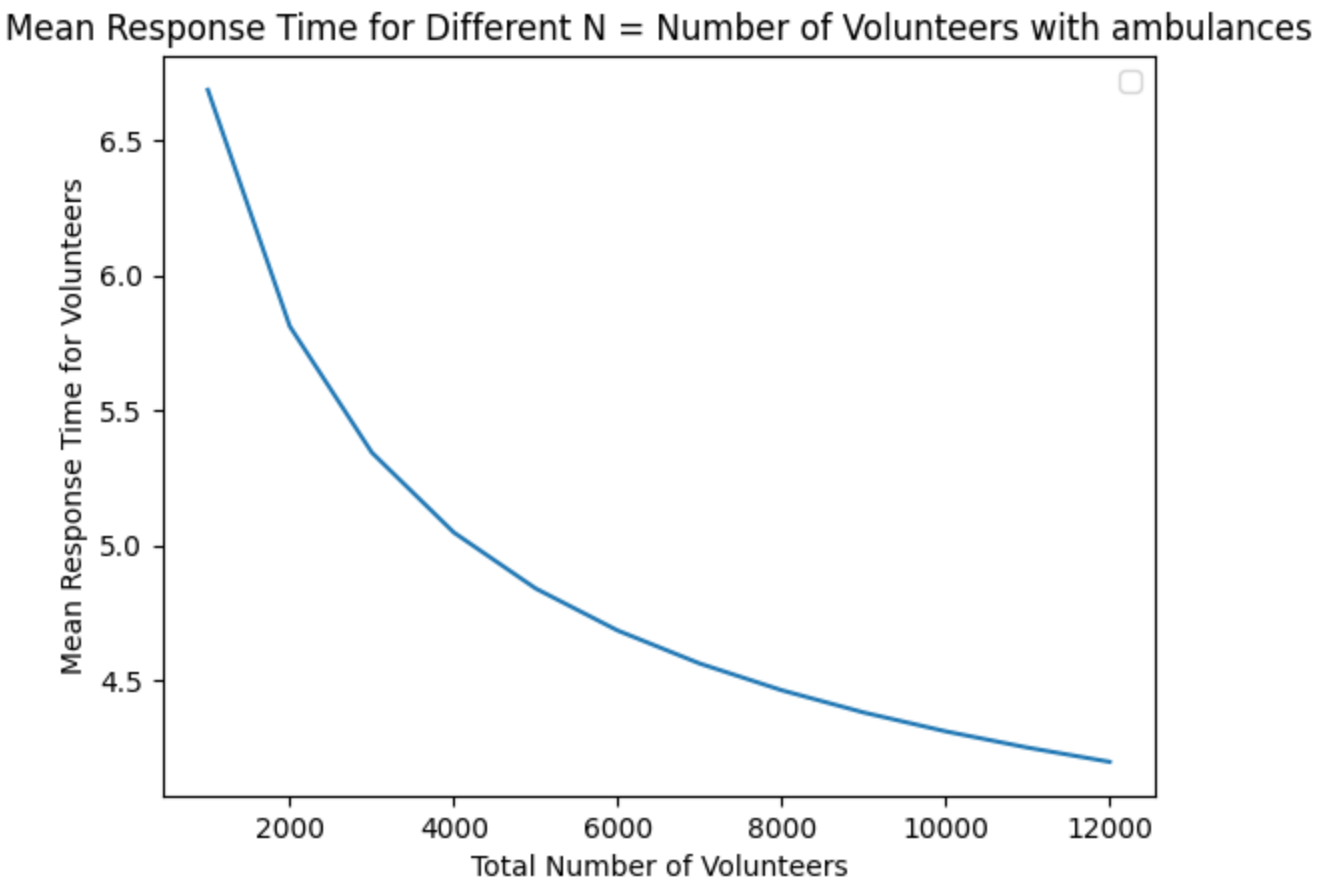


In [94]:
```
import numpy as np

# Values of N ranging from 1000 to 12001
N_values = np.arange(1000, 12001, 1000)

# Extract necessary columns from the DataFrame once
pop_sum = result['pop'].sum()
prob_ohca_values = result['probability_ohca'].values
people_per_sq_km_values = result['people_per_sq_km'].values

# Initialize an empty list to store the functions for different N values
functions = []
mean_times = []

# Pre-generate the time array
time_array = np.arange(3, 100, 0.01)

# Iterate over each N value
for N in N_values:
    function_values = []

    for t in time_array:
        exp_values = np.exp(-1 * N *.3* people_per_sq_km_values * np.pi * (0.1 ** 2) * (t - 3) *
        prob_call_values = 2*np.pi*N*.3*people_per_sq_km_values*(0.1 ** 2) * (t - 3) / pop_sum

        total_prob = np.sum(prob_ohca_values * prob_call_values * exp_values)/10
```

```
        function_values.append(total_prob)

    functions.append(function_values)

# After the loop, normalize and calculate mean_times
normalized_functions = [f/np.sum(f) for f in functions]
time_array[time_array>9]=9
mean_times = [np.dot(nf, time_array) for nf in normalized_functions]
```

In [95]:
```
plt.plot(np.arange(1000, 12001, 1000), mean_times)
plt.xlabel('Total Number of Volunteers')
plt.ylabel('Mean Response Time for Volunteers')
plt.title('Mean Response Time for Different N = Number of Volunteers with ambulances')
plt.legend()
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an unde
rscore are ignored when legend() is called with no argument.
```



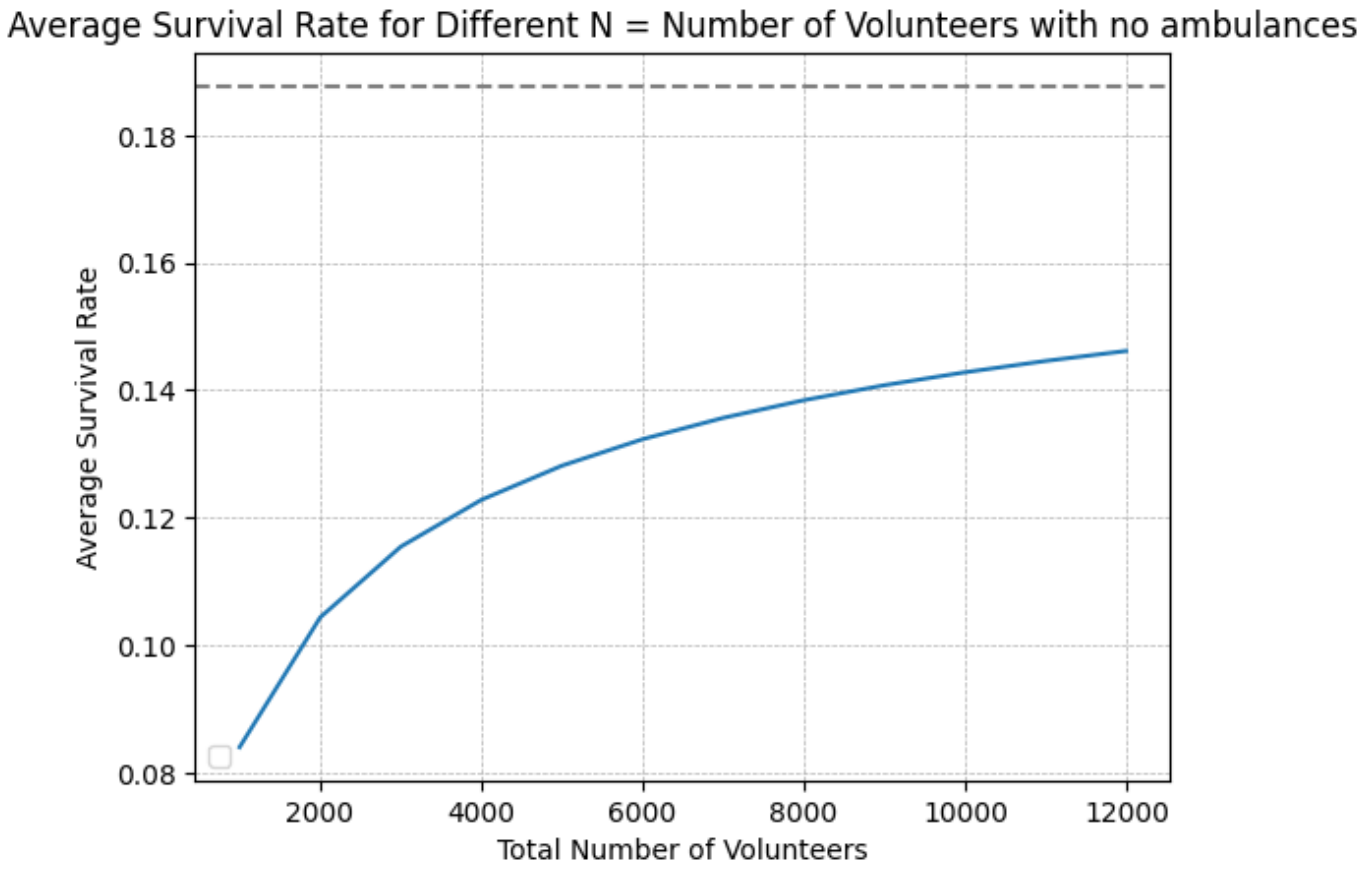
In [96]:
```
import numpy as np

# Values of N ranging from 1000 to 12001
N_values = np.arange(1000, 12001, 1000)

# Extract necessary columns from the DataFrame once
pop_sum = result['pop'].sum()
prob_ohca_values = result['probability_ohca'].values
people_per_sq_km_values = result['people_per_sq_km'].values

# Initialize an empty list to store the functions for different N values
functions = []
mean_survival = []

# Pre-generate the time array
time_array = np.arange(3, 100, 0.01)
```

```
# Iterate over each N value
for N in N_values:
    function_values = []

    for t in time_array:
        exp_values = np.exp(-1 * N *.3* people_per_sq_km_values * np.pi * (0.1 ** 2) * (t - 3) *
        prob_call_values = 2*np.pi*N*.3*people_per_sq_km_values*(0.1 ** 2) * (t - 3) / pop_sum

        total_prob = np.sum(prob_ohca_values * prob_call_values * exp_values)/10
        function_values.append(total_prob)

    functions.append(function_values)

# After the loop, normalize and calculate mean_times
normalized_functions = [f/np.sum(f) for f in functions]
mean_survival = [np.dot(nf, survival_rate(time_array)) for nf in normalized_functions]
```

In [97]:

```
plt.axhline(survival_rate(3), color='grey', linestyle='--')
plt.plot(np.arange(1000, 12001, 1000), mean_survival)
plt.xlabel('Total Number of Volunteers')
plt.ylabel('Average Survival Rate')
plt.title('Average Survival Rate for Different N = Number of Volunteers with no ambulances')
plt.legend()
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an unde
rscore are ignored when legend() is called with no argument.
```



In [103…

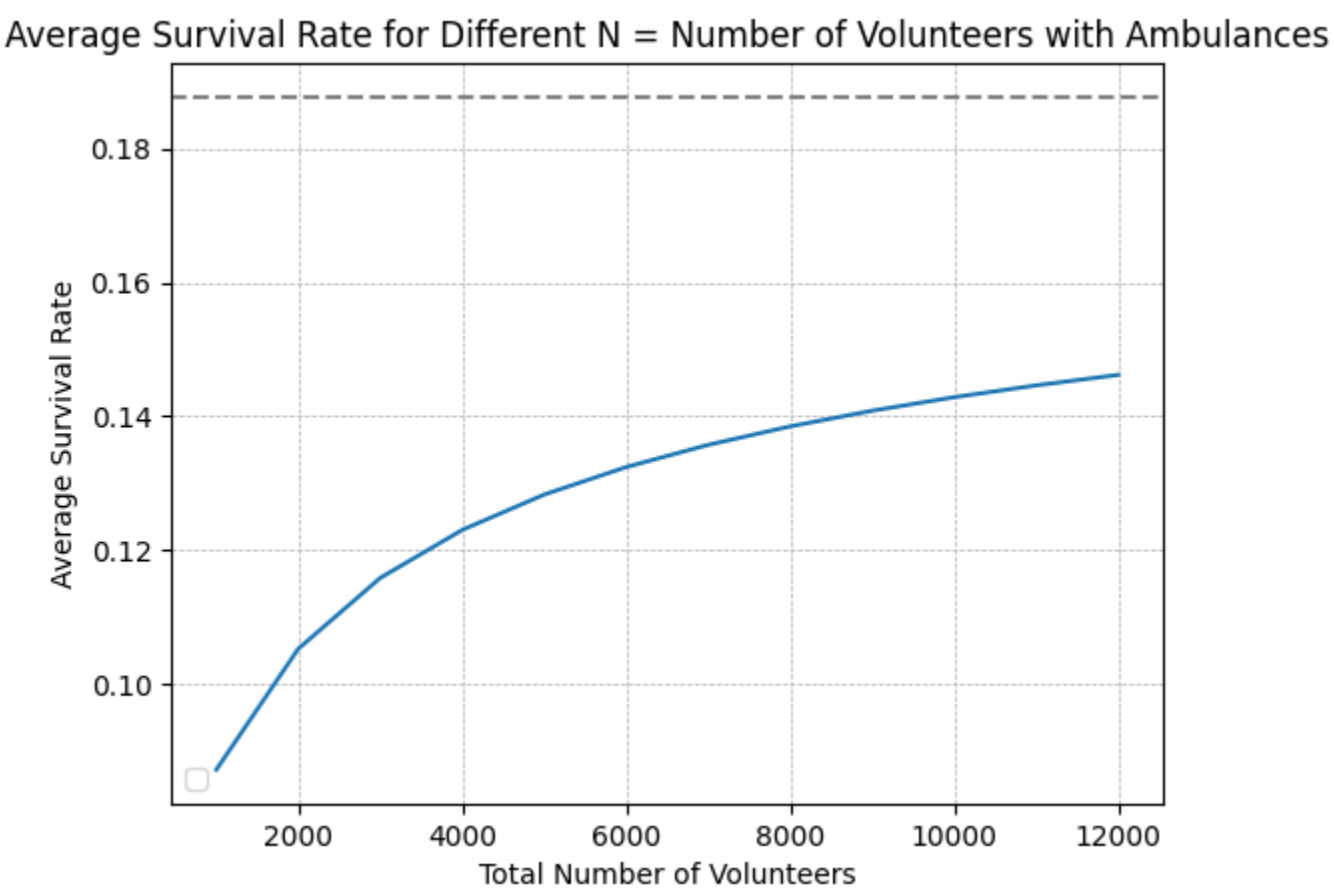
```
import numpy as np

# Values of N ranging from 1000 to 12001
N_values = np.arange(1000, 12001, 1000)

# Extract necessary columns from the DataFrame once
pop_sum = result['pop'].sum()
```

```
prob_ohca_values = result['probability_ohca'].values
people_per_sq_km_values = result['people_per_sq_km'].values

# Initialize an empty list to store the functions for different N values
functions = []
mean_survival = []

# Pre-generate the time array
time_array = np.arange(3, 100, 0.01)

# Iterate over each N value
for N in N_values:
    function_values = []

    for t in time_array:
        exp_values = np.exp(-1 * N *.3* people_per_sq_km_values * np.pi * (0.1 ** 2) * (t - 3) *
        prob_call_values = 2*np.pi*N*.3*people_per_sq_km_values*(0.1 ** 2) * (t - 3) / pop_sum

        total_prob = np.sum(prob_ohca_values * prob_call_values * exp_values)/10
        function_values.append(total_prob)

    functions.append(function_values)

# After the loop, normalize and calculate mean_times
normalized_functions = [f/np.sum(f) for f in functions]
time_array[time_array>9]=9
mean_survival = [np.dot(nf, survival_rate(time_array)) for nf in normalized_functions]
```

In [104…

```
plt.axhline(survival_rate(3), color='grey', linestyle='--')
plt.plot(np.arange(1000, 12001, 1000), mean_survival)
plt.xlabel('Total Number of Volunteers')
plt.ylabel('Average Survival Rate')
plt.title('Average Survival Rate for Different N = Number of Volunteers with Ambulances')
plt.legend()
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an unde
rscore are ignored when legend() is called with no argument.
```



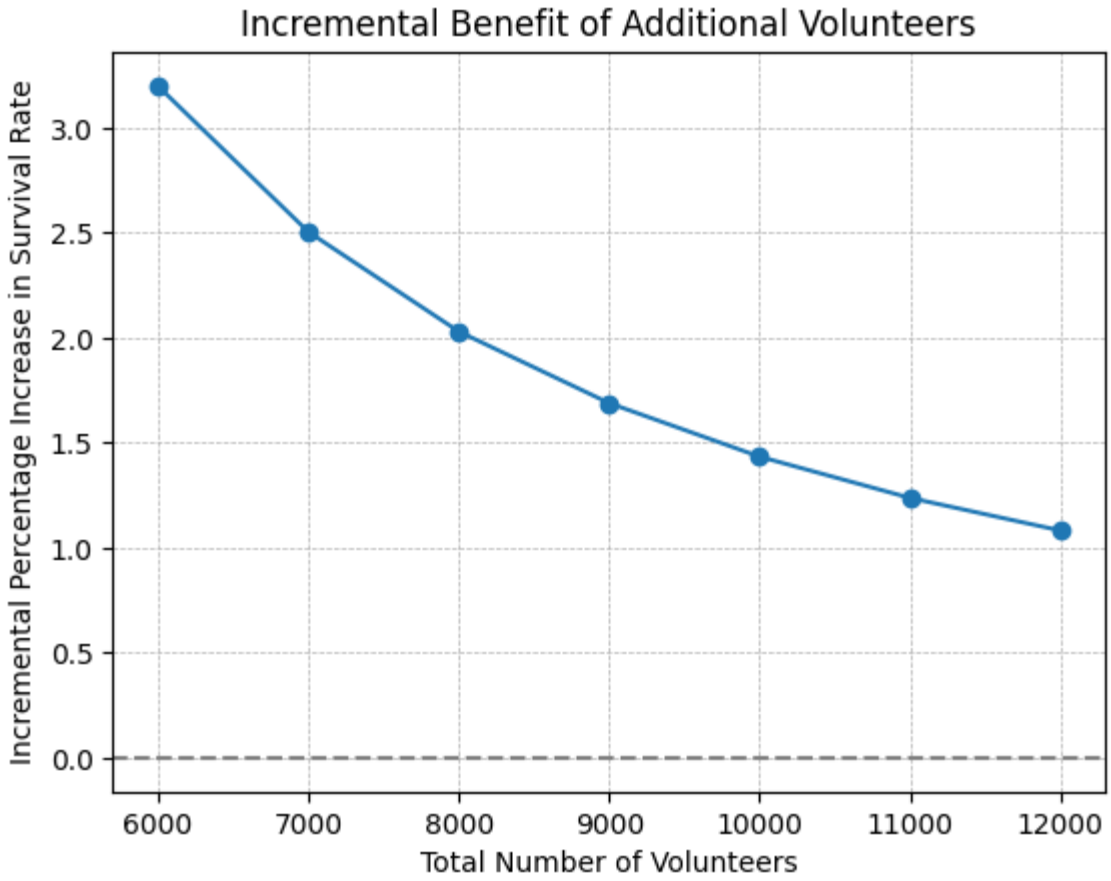
In [100…

```python
import numpy as np

# Values of N ranging from 1000 to 12001
N = 1000000

# Extract necessary columns from the DataFrame once
pop_sum = result['pop'].sum()
prob_ohca_values = result['probability_ohca'].values
people_per_sq_km_values = result['people_per_sq_km'].values

# Initialize an empty list to store the functions for different N values
functions = []
mean_survival = []

# Pre-generate the time array
time_array = np.arange(3, 100, 0.01)

function_values = []

for t in time_array:
    exp_values = np.exp(-1 * N *.3* people_per_sq_km_values * np.pi * (0.1 ** 2) * (t - 3) ** 2
    prob_call_values = 2*np.pi*N*.3*people_per_sq_km_values*(0.1 ** 2) * (t - 3) / pop_sum

    total_prob = np.sum(prob_ohca_values * prob_call_values * exp_values)/10
    function_values.append(total_prob)

functions.append(function_values)

# After the loop, normalize and calculate mean_times
normalized_functions = [f/np.sum(f) for f in functions]
mean_survival = [np.dot(nf, survival_rate(time_array)) for nf in normalized_functions]
print('Mean Survival Rate for 1,000,000 volunteers', mean_survival)
print('Max Survival Rate',survival_rate(3))
```

```
Mean Survival Rate for 1,000,000 volunteers [0.18251332253185715]
Max Survival Rate 0.1877037791895132
```

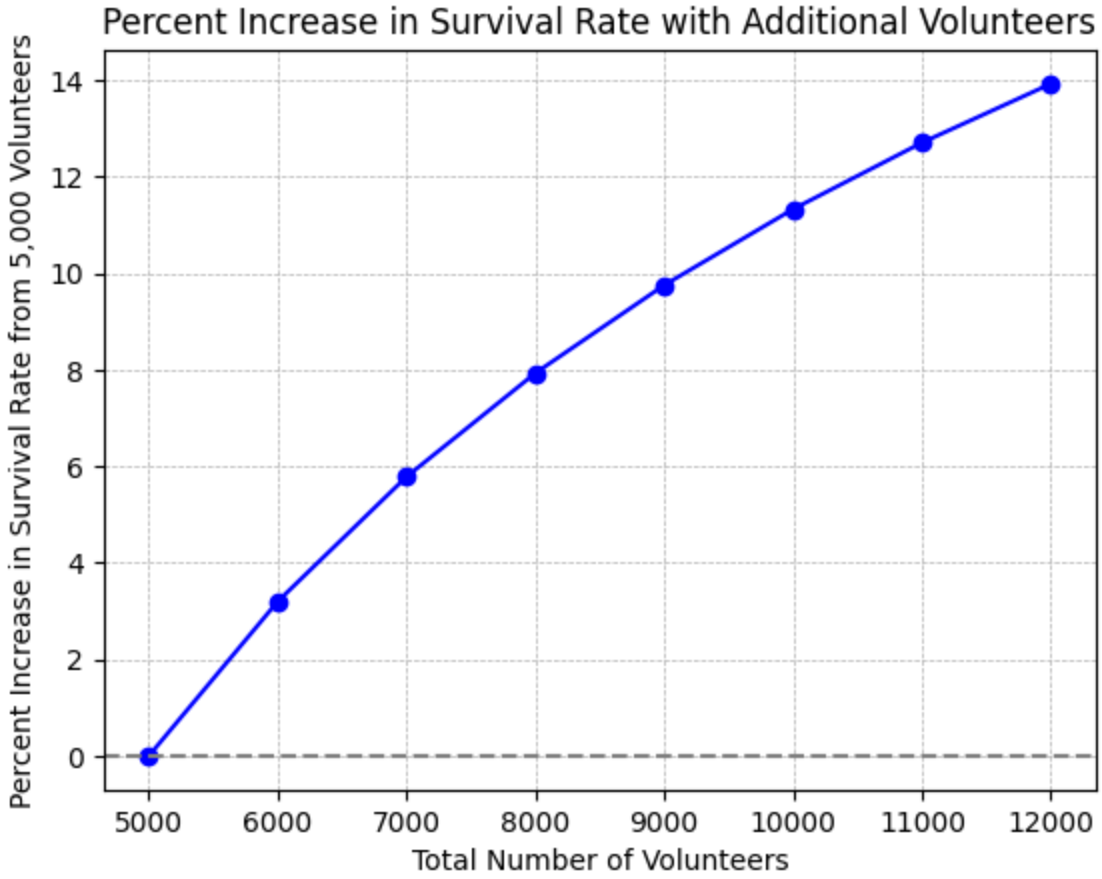
In [105…
```python
survival = mean_survival[4:]
```

In [106…
```python
#chat GPT
# First, calculate the incremental percentage increase for each step
incremental_percentage_increase = []  # start with no increase for 5000 volunteers
for i in range(1, len(survival)):
    increment = (survival[i] - survival[i-1]) / survival[i-1] * 100
    incremental_percentage_increase.append(increment)

# Then, plot this incremental percentage increase against the number of volunteers
x_values = np.arange(6000, 12001, 1000)
plt.plot(x_values, incremental_percentage_increase, marker='o')

# Enhance the plot
plt.axhline(0, color='grey', linestyle='--')  # add horizontal line at 0% for reference
plt.xlabel('Total Number of Volunteers')
plt.ylabel('Incremental Percentage Increase in Survival Rate')
plt.title('Incremental Benefit of Additional Volunteers')
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.show()
```



In [107…
```python
#chat gpt
# Find the survival rate for 5,000 volunteers
reference_survival = survival[0]  # Assuming that the first value in your survival list corresp

# Calculate the percent increase from the reference for all values
percentage_increase = [(s - reference_survival) / reference_survival * 100 for s in survival]

# Now, plot this percentage increase against the number of volunteers
x_values = np.arange(5000, 12001, 1000)
plt.plot(x_values, percentage_increase, marker='o', color='b')
```

```
# Enhance the plot
plt.axhline(0, color='grey', linestyle='--')  # add horizontal line at 0% for reference
plt.xlabel('Total Number of Volunteers')
plt.ylabel('Percent Increase in Survival Rate from 5,000 Volunteers')
plt.title('Percent Increase in Survival Rate with Additional Volunteers')
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.show()
```



In [ ]:

In addition, suppose that currently we have 5,000 volunteers in NYC allocated to locations in proportion to their population and we could recruit an additional 7,000 volunteers. We want to decide the way those volunteers are located across the set of locations in NYC in the sense of the intensity of the associated Poisson point process (this helps direct recruiting efforts). What intensity (think of this as a vector, indexed by location, that integrates to 5,000 + 7,000 = 12,000) would yield the biggest increase in survival rates? Be sure to explain how you obtained your answer and depict it graphically. (There is no need to use formal optimization methods, but at the very least you should explain how you obtained your answer and report its impact on the survival rate. Ideally you would have performed some simple mathematical modeling to gain some insight into how to allocate volunteers to locations.) You might provide a scatterplot where each point corresponds to a single location, with population density on the x axis and your suggested volunteer density on the y axis and discuss what your scatter plot shows.

We see that we want the maximum number of volunteers to maximize survival. Regarding the allocation, I will optimize by maximizing the probability of survival, which is the same as minimizing the probability of death.

Breaking up the possible locations of volunteers into the 1163 nodes, it has been shown that the probability of death is a function of the response time.

let $p = f(T_v) \in [0,1]$ be the probability of death assuming that the volunteer response time is T_v.

This probability is minimized when T_v=3, since this is the minimum response time of volunteers. We assume that the probability of death is an increasing continuous function. We will set the minimum probability to be d_{min}. The maximum probability of death will be assuming that the response time is infinite. We define this as d_{max}

Since the response time is dependent on the probability of OHCA and the location of volunteers. We know that $T_v = f(v)$ where v is the distribution of volunteers across the city.

We will assume that the first 5,000 volunteers are distributed according to the population across the city. These volunteers will be distributed according to a spacial poisson process.

The death probability as a function of v, the volunteer distribution, is

$$d(v) = E[f(T_v)] = \int_0^1 P(f(T_v) > u)du = d_{min} + \int_{d_{min}}^{d_{max}} P(T_v > f^{-1}(u))du$$

This then needs to be split up based on the location of the call.

$$d(v) = d_{min} + \sum_l \lambda_l \int_{d_{min}}^{d_{max}} P(T_v > f^{-1}(u))du$$

Where λ_l is the probability that a call originates in node l

We will use a greedy approach in which volunteers are added to locations where they would yield the highest marginal reduction in death rate. We will allocate volunteers in groups of one volunteer.

Allocation of first 5000 volunteers

In [180…
```
result['allocation_5000'] = 5000 * result['volunteer_scale']
```

In [181…
```
def greedy_algo(result,v):
    # v is expected number of volunteers at location i for 5000 volunteers
    survival = [0] * len(result)
    added_survival = [0] * len(result)
    N = 5000
    for i in range(len(result)):
        survival[i] = evaluate_survival(i,v[i], N)
        added_survival[i] = evaluate_survival(i,v[i]+1, N)
    for i in range(5000,12000):
        N = i
        added_location = np.argmax(added_survival)
        v[added_location] += 1
        survival[added_location] += added_survival[added_location]
        added_survival[added_location] = evaluate_survival(added_location,v[added_location]+1, N
    return v

def evaluate_survival(location,v, N):
    volunteer_scale = v/N
    people_per_sq_km = volunteer_scale*sum(result['people_per_sq_km'])
    prob_ohca_values = result['probability_ohca'][location]
    pop_sum = result['pop'].sum()
    functions = []
    survival = []
    time_array = np.arange(3,20,.01)
    for t in time_array:
        exp_values = np.exp(-1 * N *.3* people_per_sq_km * np.pi * (0.1 ** 2) * (t - 3) ** 2 / p
```

```
        prob_call_values = 2*np.pi*N*.3*people_per_sq_km*(0.1 ** 2) * (t - 3) / pop_sum
        total_prob = prob_call_values * exp_values
        functions.append(total_prob)
    normalized_functions = [f/np.sum(functions) for f in functions]
    survival_values = [nf * survival_rate(t) for nf, t in zip(normalized_functions, time_array)]
    return sum(survival_values)*prob_ohca_values
```

In [182…

```
v = greedy_algo(result,result['allocation_5000'])
```

```
C:\Users\julia\anaconda3\envs\engri_1101\lib\site-packages\ipykernel_launcher.py:12: SettingWith
CopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/in
dexing.html#returning-a-view-versus-a-copy
  if sys.path[0] == "":
```

In [192…

```
N = 12000

# Extract necessary columns from the DataFrame once
pop_sum = result['pop'].sum()
prob_ohca_values = result['probability_ohca'].values
people_per_sq_km = v*sum(result['people_per_sq_km'])

# Initialize an empty list to store the functions for different N values
functions = []
mean_survival = []
time_array = np.arange(3, 100, 0.01)

for t in time_array:
    exp_values = np.exp(-1 * N *.3* people_per_sq_km * np.pi * (0.1 ** 2) * (t - 3) ** 2 / pop_s
    prob_call_values = 2*np.pi*N*.3*people_per_sq_km*(0.1 ** 2) * (t - 3) / pop_sum

    total_prob = np.sum(prob_ohca_values * prob_call_values * exp_values)/10
    functions.append(total_prob)

# After the loop, normalize and calculate mean_times
normalized_functions = [f/np.sum(functions) for f in functions]
survival_values = [nf * survival_rate(t) for nf, t in zip(normalized_functions, time_array)]
print('Survival Rate for Optimal Allocation with 12,000 Volunteers', sum(survival_values))
```

```
Survival Rate for Optimal Allocation with 12,000 Volunteers 0.18706721586702604
```

In [195…

```
survival_rate(3)
```

Out[195]:

```
0.1877037791895132
```

In [184…

```
N = 12000

# Extract necessary columns from the DataFrame once
pop_sum = result['pop'].sum()
prob_ohca_values = result['probability_ohca'].values
people_per_sq_km_values = result['people_per_sq_km'].values

# Initialize an empty list to store the functions for different N values
functions = []
mean_survival = []
time_array = np.arange(3, 100, 0.01)

for t in time_array:
    exp_values = np.exp(-1 * N *.3* people_per_sq_km_values * np.pi * (0.1 ** 2) * (t - 3) ** 2
    prob_call_values = 2*np.pi*N*.3*people_per_sq_km_values*(0.1 ** 2) * (t - 3) / pop_sum

    total_prob = np.sum(prob_ohca_values * prob_call_values * exp_values)/10
```

```
    functions.append(total_prob)

# After the loop, normalize and calculate mean_times
normalized_functions = [f/np.sum(functions) for f in functions]
survival_values = [nf * survival_rate(t) for nf, t in zip(normalized_functions, time_array)]
print('Survival Rate for Population Proportional Allocation with 12,000 Volunteers', sum(surviva
```

```
Survival Rate for Population Proportional Allocation with 12,000 Volunteers 0.14615936874294902
```

In [196…

```
def generate_OHCA():
    nodes = result['name']
    probability = result['probability_ohca']
    node_OHCA= np.random.choice(nodes,p=probability)
    return result[result['name'] == node_OHCA]
```

In [204…

```
# Written by Chat GPT

def generate_volunteers_with_coordinates(N, v):
    # Calculate the maximum population density among all nodes

    people_per_sq_km = v*sum(result['people_per_sq_km'])

    max_density = people_per_sq_km.max()

    # Initialize a dictionary to store the number of volunteers for each node
    node_volunteer_count = {node_name: 0 for node_name in result['name']}

    # Set the desired total number of volunteers
    total_volunteers = N

    # Initialize lists to store volunteer coordinates
    volunteer_coordinates = []

    # Loop to generate volunteers
    while sum(node_volunteer_count.values()) < total_volunteers:
        node = np.random.choice(result['name'], p=result['probability_ohca'])
        lon = float(result[result['name'] == node]['lon'])
        lat = float(result[result['name'] == node]['lat'])

        # Assuming 'node' contains the name of the closest node
        closest_node_index = result[result['name'] == node].index[0]

        # Get the closest node
        closest_node = result.iloc[closest_node_index]

        # Calculate the probability of selecting this location based on population density
        probability = people_per_sq_km[closest_node_index] / max_density

        # Generate a random number between 0 and 1
        random_number = np.random.uniform(0, 1)

        # Accept the location if random_number < probability
        if random_number < probability:
            # Update the count of volunteers for the selected node
            node_volunteer_count[closest_node['name']] += 1
            print(sum(node_volunteer_count.values()))

            # Generate coordinates within a circle around the volunteer location
            center_lat = closest_node['lat']
            center_lon = closest_node['lon']
            radius_meters = np.sqrt(closest_node['area (m^2)\r'] / np.pi)
            num_points = 1  # One volunteer per location

            # Earth's radius in meters
```

```
            earth_radius = 6371000.0  # Approximate value for Earth's radius

            # Generate random radii and angles for each point
            radii = np.sqrt(np.random.uniform(0, 1, num_points)) * radius_meters
            angles = np.random.uniform(0, 2 * np.pi, num_points)

            # Calculate latitude and longitude offsets for each point
            lat_offsets = radii * np.sin(angles) / earth_radius
            lon_offsets = radii * np.cos(angles) / (earth_radius * np.cos(np.radians(center_lat)

            # Calculate the final latitude and longitude coordinates
            latitudes = center_lat + np.degrees(lat_offsets)
            longitudes = center_lon + np.degrees(lon_offsets)

            # Append the coordinates to the volunteer_coordinates list
            volunteer_coordinates.append({'longitude': longitudes[0], 'latitude': latitudes[0]})

    # Create a DataFrame with the node name and the number of volunteers
    volunteer_count_df = pd.DataFrame({'node_name': list(node_volunteer_count.keys()),
                                       'number_of_volunteers': list(node_volunteer_count.values(

    # Create a DataFrame with the volunteer coordinates
    volunteer_coordinate_df = pd.DataFrame(volunteer_coordinates)

    return volunteer_count_df, volunteer_coordinate_df
```

For each OHCA, available volunteers around OHCA is possion with mean

In [ ]:
```
volunteer_count_df, volunteer_coordinate_df = generate_volunteers_with_coordinates(12000,v)
```

In [206…
```
import numpy as np
def merc_from_arrays(lats, lons):
    r_major = 6378137.000
    x = r_major * np.radians(lons)
    scale = x/lons
    y = 180.0/np.pi * np.log(np.tan(np.pi/4.0 + lats * (np.pi/180.0)/2.0)) * scale
    return (x, y)

volunteer_coordinate_df['x'],volunteer_coordinate_df['y'] = merc_from_arrays(volunteer_coordinat
```

In [207…
```
output_notebook()
def plotNetwork(nodes, links, dis_time, criteria, population, volunteer_coordinate_df, title='Pl
    """
    Plots a static map of a network = (nodes, links).
    :param nodes: pandas df with 'name', 'x', 'y' cols
                  'x' and 'y' treated as mercator coords
    :param links: pandas df with 'start' and 'end' cols
                  with entries matching nodes' names
    :param title: str of graph title
    :param target: list of node names
    :param on_map: boolean for map background
    """

    nodes = nodes
    links = links
    dfs = dis_time
    population = population
    speed = dfs['distance'].values/dfs['time'].values.tolist()
    volunteers = volunteer_coordinate_df
    node_ids_v = volunteers.index.values.tolist()
    x_v = volunteers.x.values.tolist()
    y_v = volunteers.y.values.tolist()
```

```
if len(criteria)>0:
    groups = len(criteria)

    color = cm.get_cmap('Blues', groups-2)    # PiYG
    c_map = {}
    c_map[0] = '#ff0000'
    #c_map[1] = '#99c199'
    for i in range(color.N):
        rgba = color(i)
        # rgb2hex accepts rgb or rgba
        c_map[i+1] = matplotlib.colors.rgb2hex(rgba)
    c_map[i+2] = '#FFFF00'


    e_clr = []
    for i in speed:
        for k in range(groups):
            if i>=criteria[k][0] and i<=criteria[k][1]:
                e_clr.append(c_map[k])
                break

else:
    e_clr = []
    for i in speed:
            e_clr.append('#A0A0A0')


# extract data
node_ids = nodes.name.values.tolist()
start = links.start.values.tolist()
end = links.end.values.tolist()
x = nodes.x.values.tolist()
y = nodes.y.values.tolist()


# get plot boundaries
min_x, max_x = min(nodes.x)+2000, max(nodes.x)-2000
min_y, max_y = min(nodes.y)+2000, max(nodes.y)-2000

plot = figure(x_range=(min_x, max_x), y_range=(min_y, max_y),
              x_axis_type="mercator", y_axis_type="mercator",
              title=title,
              width=600, height=470,
              toolbar_location=None, tools=[]
             )

graph = GraphRenderer()

if on_map == True:
    # add map tile
    plot.add_tile(get_provider(Vendors.CARTODBPOSITRON_RETINA))

# define nodes
graph.node_renderer.data_source.add(node_ids, 'index')
graph.node_renderer.data_source.add(node_ids_v, 'index_v')
graph.node_renderer.glyph = Circle(line_color='green', line_alpha=0,
                                   fill_color='green', size=3.5,
                                   fill_alpha=0
                                  )



# define edges
# graph.edge_renderer.data_source.data = dict(start=list(start),
```

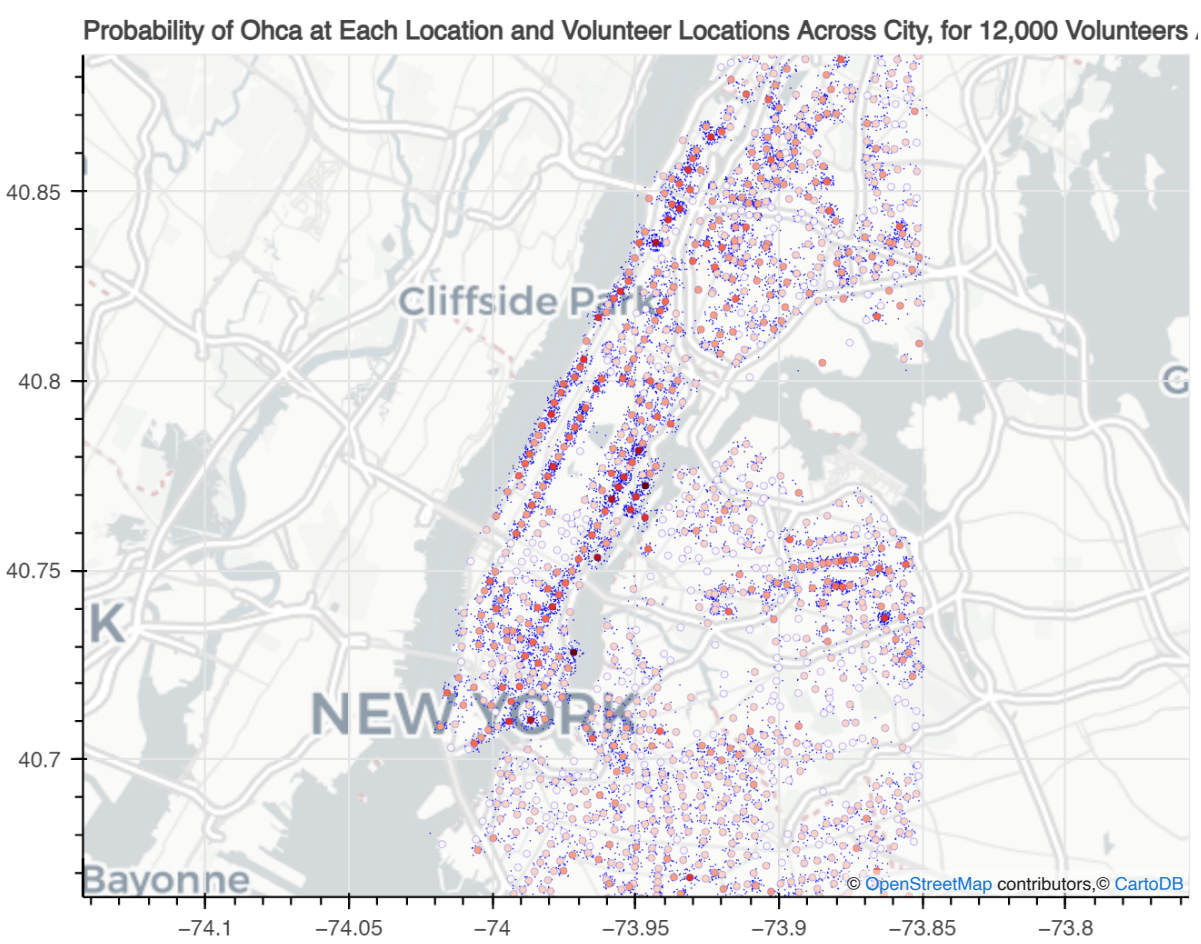
```
    #                                          end=list(end),e_clr=e_clr
    #                                         )
    # graph.edge_renderer.glyph = MultiLine(line_color = 'e_clr',
    #                                       line_alpha=1, line_width=.6
    #                                      )

    # set node locations
    graph_layout = dict(zip(node_ids, zip(x, y)))
    graph.layout_provider = StaticLayoutProvider(graph_layout=graph_layout)

    plot.renderers.append(graph)

    source_v = ColumnDataSource(volunteers)
    vols = Circle(x="x", y="y", size=.5, line_color='blue',
                        fill_color='blue', line_width=0.1
                       )
    plot.add_glyph(source_v, vols)


    # add POIS
    if len(population)>0:
        color = cm.get_cmap('Reds', 7)
        for i in range(7):
            target = []
            for index,row in population.iterrows():
                if row['pop']>=(i*2500) and row['pop']<((i+1)*2500):
                    target.append(row['name'])

            populationp = nodes.loc[nodes['name'].isin(target)]
            source = ColumnDataSource(populationp)
            poi = Circle(x="x", y="y", size=3.5, line_color='blue',
                        fill_color=matplotlib.colors.rgb2hex(color(i)), line_width=0.1
                       )

            plot.add_glyph(source, poi)

    show(plot)
```

BokehJS 2.4.3 successfully loaded.

In [208…

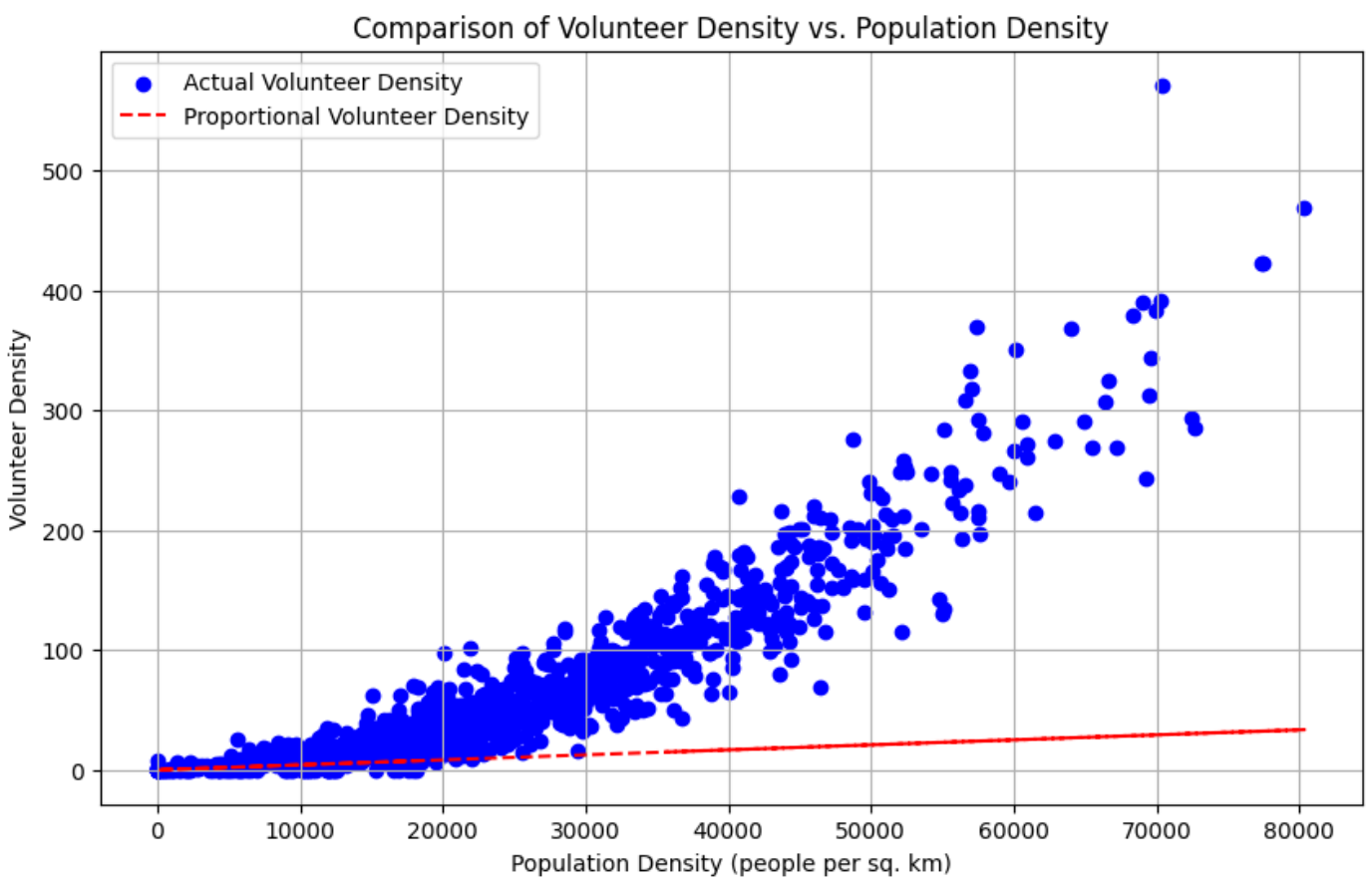
```
plotNetwork(nodes, links,dfs, [], population, volunteer_coordinate_df, title="Probability of Ohc
```

```
BokehUserWarning: ColumnDataSource's columns must be of the same length. Current lengths: ('inde
x', 20056), ('index_v', 12000)
```



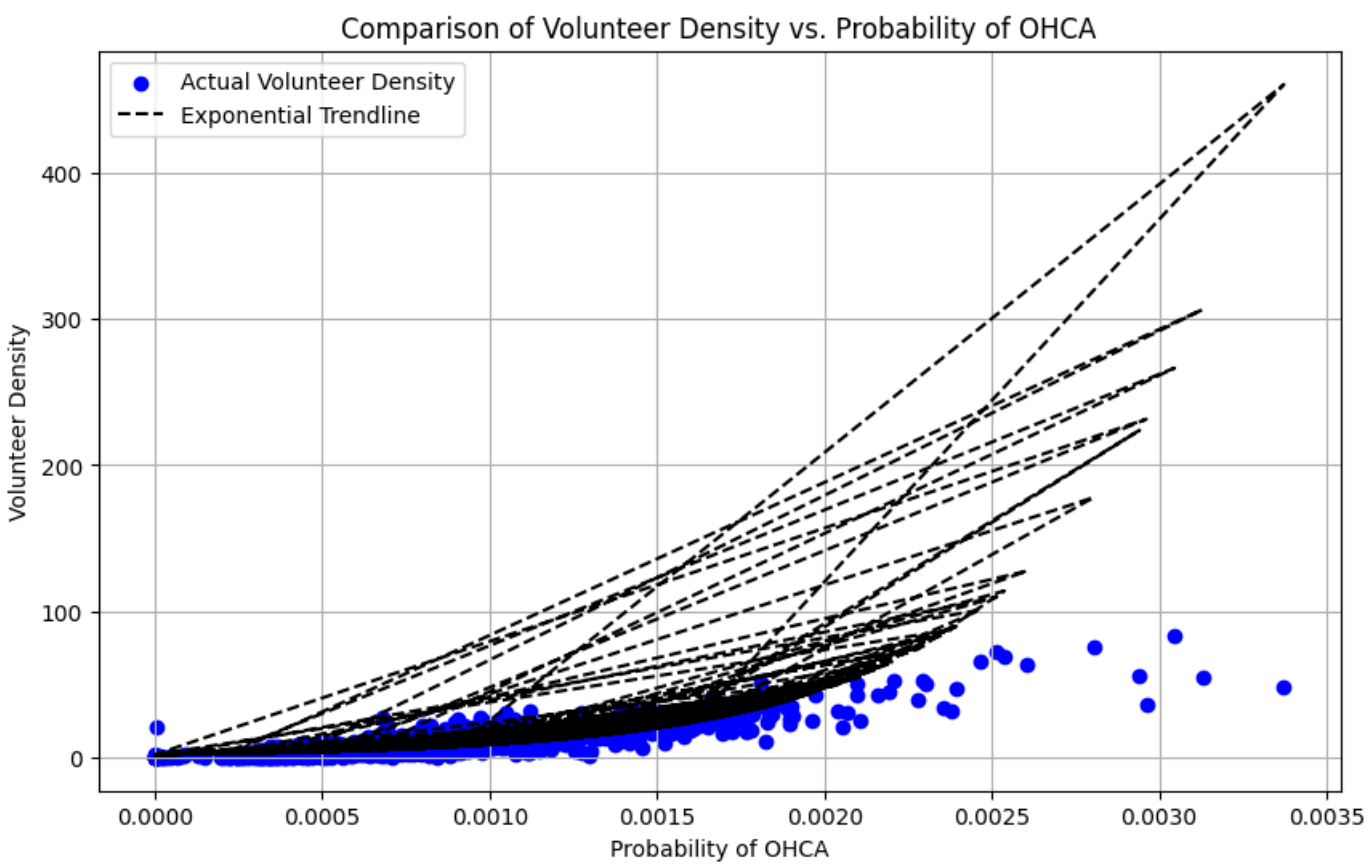
In [278…

```
#CHAT GPT

total_volunteers = sum(volunteer_count_df['number_of_volunteers'])
population_density = result['people_per_sq_km']
total_population_density = sum(population_density)
proportional_v = [total_volunteers * (density / total_population_density) for density in populat

# Plot
plt.figure(figsize=(10, 6))
plt.scatter(population_density, volunteer_count_df['number_of_volunteers']/result['area km^2'],
plt.plot(population_density, total_volunteers*result['people_per_sq_km']/sum(result['people_per_
plt.xlabel('Population Density (people per sq. km)')
plt.ylabel('Volunteer Density')
plt.title('Comparison of Volunteer Density vs. Population Density')
plt.legend()
plt.grid(True)
plt.show()
```



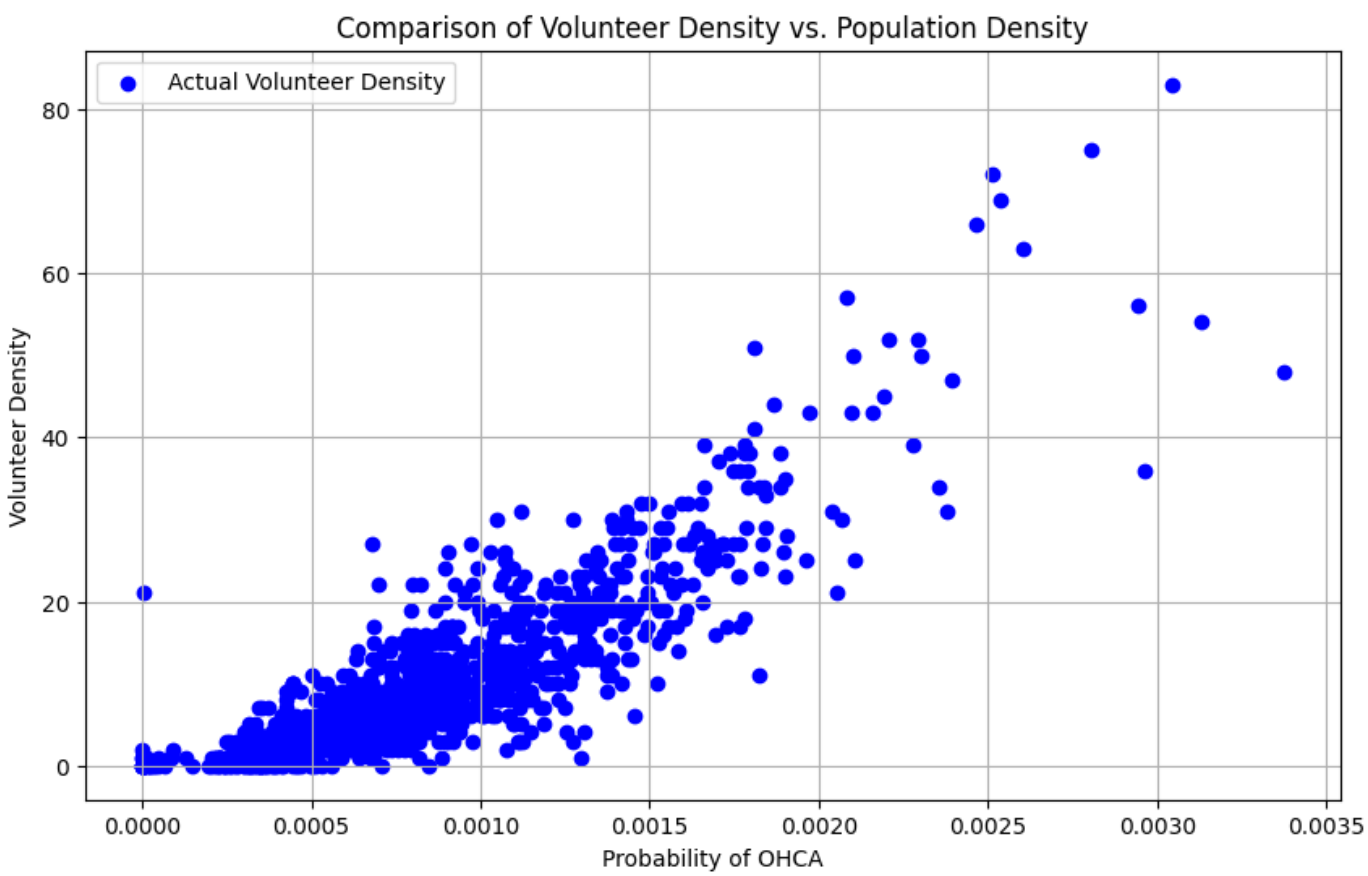
In [286…

```
#CHAT GPT

# Given data
probability_ohca = result['probability_ohca']
volunteer_density = volunteer_count_df['number_of_volunteers']

# Linear regression on the logarithm of the volunteer_density
log_volunteer_density = np.log(volunteer_density[volunteer_density>0])
z = np.polyfit(probability_ohca[volunteer_density>0], log_volunteer_density, 1)
p = np.poly1d(z)

# Calculate exponential trendline values
trendline = np.exp(p(probability_ohca))

# Plot
plt.figure(figsize=(10, 6))

# Scatter plot for actual volunteer density
plt.scatter(probability_ohca, volunteer_density, color='blue', label='Actual Volunteer Density')

# Plot exponential trendline
plt.plot(probability_ohca, trendline, "k--", label='Exponential Trendline')

plt.xlabel('Probability of OHCA')
plt.ylabel('Volunteer Density')
plt.title('Comparison of Volunteer Density vs. Probability of OHCA')
plt.legend()
plt.grid(True)
plt.show()
```



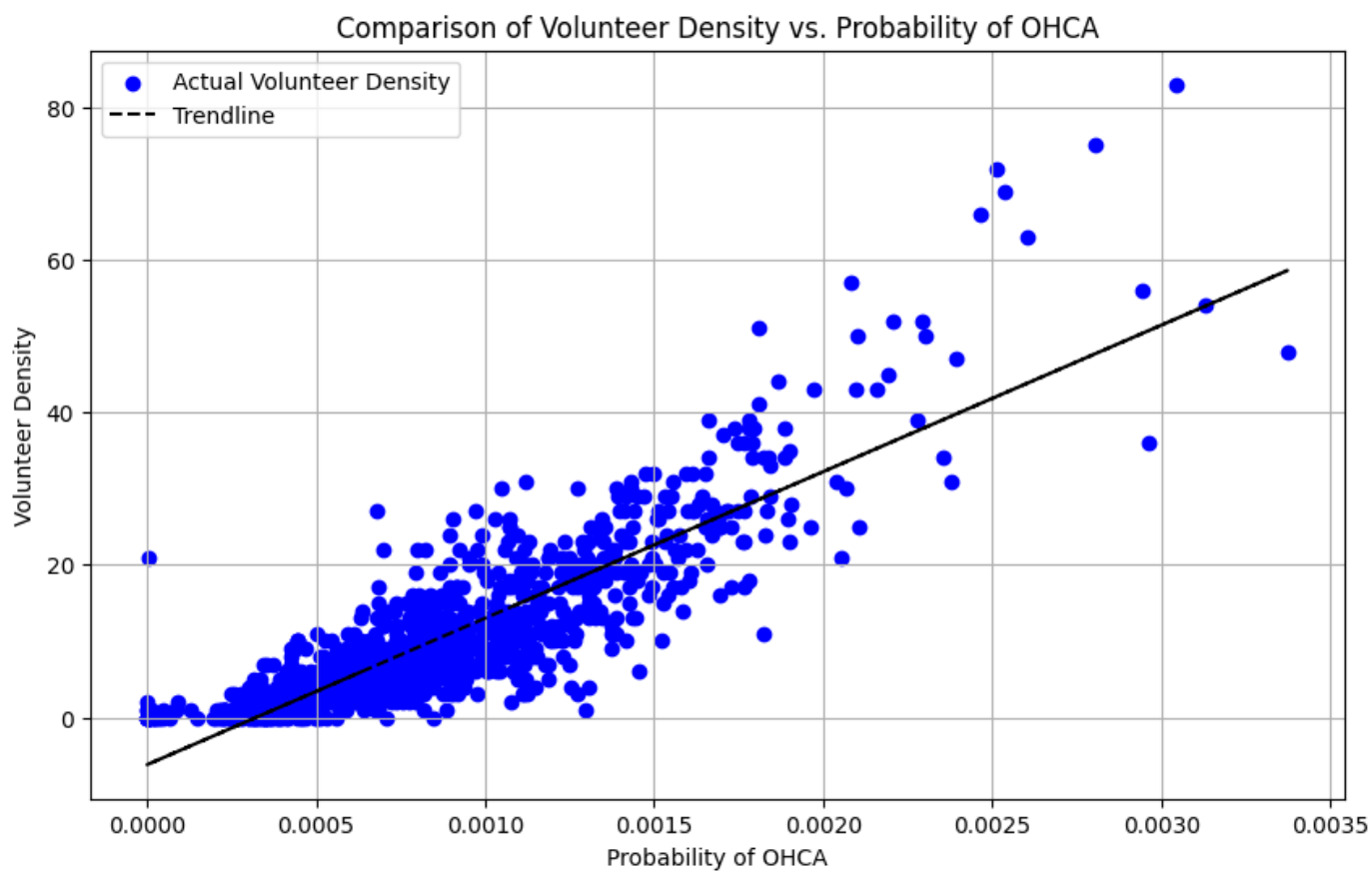
In [273…

```
#CHAT GPT

# Plot
plt.figure(figsize=(10, 6))
plt.scatter(result['probability_ohca'], volunteer_count_df['number_of_volunteers'], color='blue'
plt.xlabel('Probability of OHCA')
plt.ylabel('Volunteer Density')
plt.title('Comparison of Volunteer Density vs. Population Density')
plt.legend()
plt.grid(True)
plt.show()
```



In [283…

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

# Given data
probability_ohca = result['probability_ohca']
volunteer_density = volunteer_count_df['number_of_volunteers']

# Calculate trendline
z = np.polyfit(probability_ohca, volunteer_density, 1)
p = np.poly1d(z)

# Plot
plt.figure(figsize=(10, 6))

# Scatter plot for actual volunteer density
plt.scatter(probability_ohca, volunteer_density, color='blue', label='Actual Volunteer Density')

# Plot trendline
plt.plot(probability_ohca, p(probability_ohca), "k--", label='Trendline')

plt.xlabel('Probability of OHCA')
plt.ylabel('Volunteer Density')
plt.title('Comparison of Volunteer Density vs. Probability of OHCA')
plt.legend()
plt.grid(True)
plt.show()
```


Comparison of Volunteer Density vs. Probability of OHCA
Actual Volunteer Density
Trendline
Volunteer Density
Probability of OHCA
0
20
40
60
80
0.0000
0.0005
0.0010
0.0015
0.0020
0.0025
0.0030
0.0035

In [ ]: