HI! If you're unfamiliar with Colab here's how to start:

1. Create your own copy of this file by clicking `File → Save A Copy In Drive`
2. Follow the prompt to sign into your Google account if necessary to open the new file in your own GDrive. Once you're in your own copy you will be able to make edits, which will allow you to use the sliders on the graphs.
3. Run all of the cells by clicking `Runtime → Run All`
4. As stated below, the first cell will prompt you to `Restart Session` to allow some installed files to take effect. Click restart session and/or refresh the page. You may need to repeat step 3.
5. Feel free to look around, but the two cells with graphs of the results are together at the very bottom. Play around with the sliders to see all of the data. I've noticed that very rarely the sliders happen to disappear for some reason. If that happens you can click `shift` + `enter` on the code line above it to refresh the graph and view the sliders.

there's an issue with the newest versions of plotly interacting with the interactive features of ipywidget. You need to run this cell and restart the colab session before running all of the others

[1] 55s
```
pip install plotly==5.10
```
```
Collecting plotly==5.10
  Downloading plotly-5.10.0-py2.py3-none-any.whl.metadata (6.9 kB)
Requirement already satisfied: tenacity>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from plotly==5.10) (9.0.0)
Downloading plotly-5.10.0-py2.py3-none-any.whl (15.2 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 15.2/15.2 MB 18.2 MB/s eta 0:00:00
Installing collected packages: plotly
  Attempting uninstall: plotly
    Found existing installation: plotly 5.24.1
    Uninstalling plotly-5.24.1:
      Successfully uninstalled plotly-5.24.1
Successfully installed plotly-5.10.0
```

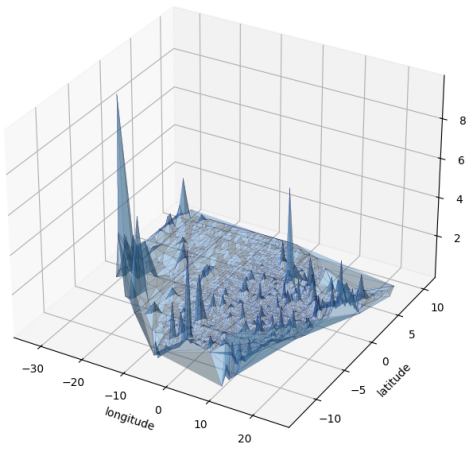
[2] 11s
```
import pandas as pd
import numpy as np
import torch
from torch import nn
import matplotlib.pyplot as plt
import os
from ipywidgets import interact, interactive
import ipywidgets as widgets
from scipy import stats

print(torch.__version__)
```
```
2.5.1+cu121
```

[3] 0s
```
# cheating in some variables here from some other data processing I did in a different project
MEAN_LAT = 38.28132872306216
MEAN_LNG = -91.65855796075903
```

[4] 1s
```
# copying in the draw counties method for our visualizations
drawing_data = 'https://www.tyro.work/cont-us-counties-latlng-normalized.csv'
if not os.path.exists('/content/cont-us-counties-latlng-normalized.csv'):
  !wget $drawing_data
else:
  print("County Location Data has already been downloaded.")

drawing_df = pd.read_csv('/content/cont-us-counties-latlng-normalized.csv')
draw_lat_tensor = torch.tensor(drawing_df['lat'].values)
draw_lat_tensor = draw_lat_tensor.reshape(-1, 1)
draw_lng_tensor = torch.tensor(drawing_df['lng'].values)
draw_lng_tensor = draw_lng_tensor.reshape(-1, 1)
# i thought we needed to vstack these, but we actually need hstack
draw_lnglat_tensor = torch.hstack((draw_lng_tensor, draw_lat_tensor))
# print("county data shape: ", draw_lnglat_tensor.shape)
# print('county data: ', draw_lnglat_tensor)
# these constants will be useful for randomizing our centroids
lat_max = draw_lat_tensor.max().item()
lat_min = draw_lat_tensor.min().item()
lng_max = draw_lng_tensor.max().item()
lng_min = draw_lng_tensor.min().item()

def draw_counties():
  plt.figure(figsize = (8,5))
  plt.scatter(draw_lnglat_tensor[:, 0], draw_lnglat_tensor[:, 1], c='#79d2a4', marker='.', s=4, label='U.S. Counties')
  plt.xlabel('Longitude')
  plt.ylabel('Latitude')
  # Instead of converting all the data back to normal latitudes and longitudes, I'm gonna just hack the axis x_labels
  x_locs, x_labels = plt.xticks()
  x_labels = [int(item + MEAN_LNG) for item in x_locs]
  plt.xticks(x_locs, x_labels)
  y_locs, y_labels = plt.yticks()
  y_labels = [int(item + MEAN_LAT) for item in y_locs]
  plt.yticks(y_locs, y_labels)

# now we have a method to quickly draw the county locations
# draw_counties()
```
```
--2024-12-02 06:54:10--  https://www.tyro.work/cont-us-counties-latlng-normalized.csv
Resolving www.tyro.work (www.tyro.work)... 162.244.93.7, 2602:faa9:1008:918:fd16:9073:7cff:b175
Connecting to www.tyro.work (www.tyro.work)|162.244.93.7|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 306413 (299K) [text/csv]
Saving to: ‘cont-us-counties-latlng-normalized.csv’

cont-us-counties-la 100%[===================>] 299.23K  1.36MB/s    in 0.2s

2024-12-02 06:54:10 (1.36 MB/s) - ‘cont-us-counties-latlng-normalized.csv’ saved [306413/306413]
```

[5] 3s
```
# from the old work I did on https://github.com/Tyrowo/us-county-data-processing-and-visualization/blob/main/data%20visualization.py
# just using the current files instead. Unfortunately 3dmatplotlib plots don't work on Colab, but I've included it just because it looks nice.

y = drawing_df['lat'].to_list()
x = drawing_df['lng'].to_list()
z = drawing_df['population'].to_list()
# lat: [25.3192, 48.8259] and lng: [-124.1568, -67.6287]
lat_dist = 48.8259 - 25.3192
lng_dist = 124.1568 - 67.6287

fig = plt.figure(figsize=(12, 8))
# plt.gca().set_aspect('equal')
ax = fig.add_subplot(111, projection='3d')
# #56a0d3 = carolina blue, #000080 = navy, alpha is transparency, linewidths should be small because there's a lot
ax.plot_trisurf(x,y,z, color='#56a0d3', edgecolors='#000080', alpha=0.4, linewidths=0.05)
# ax.scatter(x,y,z, c='white', alpha = 0.05, marker='.')
# Add labels and title
ax.set_xlabel('longitude')
ax.set_ylabel('latitude')
ax.set_zlabel('population')
# plt.plot(x, y, markersize=2, linestyle='None')
```

```
plt.show()
```



[6]
```
model_pytorch = 'https://www.tyro.work/optimized_model.pth'
if not os.path.exists('/content/optimized_model.pth'):
  !wget $model_pytorch
else:
  print("Model has already been downloaded.")

k_means_pytorch = 'https://www.tyro.work/k_means_model_data.pth'
if not os.path.exists('/content/k_means_model_data.pth'):
  !wget $k_means_pytorch
else:
  print("K Means Data has already been downloaded.")
```

```
--2024-12-02 06:54:14--  https://www.tyro.work/optimized_model.pth
Resolving www.tyro.work (www.tyro.work)... 162.244.93.7, 2602:faa9:1008:918:fd16:9073:7cff:b175
Connecting to www.tyro.work (www.tyro.work)|162.244.93.7|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1882670 (1.8M) [application/octet-stream]
Saving to: ‘optimized_model.pth’

optimized_model.pth 100%[===================>]   1.79M  4.99MB/s    in 0.4s

2024-12-02 06:54:14 (4.99 MB/s) - ‘optimized_model.pth’ saved [1882670/1882670]

--2024-12-02 06:54:14--  https://www.tyro.work/k_means_model_data.pth
Resolving www.tyro.work (www.tyro.work)... 162.244.93.7, 2602:faa9:1008:918:fd16:9073:7cff:b175
Connecting to www.tyro.work (www.tyro.work)|162.244.93.7|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2106323 (2.0M) [application/octet-stream]
Saving to: ‘k_means_model_data.pth’

k_means_model_data. 100%[===================>]   2.01M  5.52MB/s    in 0.4s

2024-12-02 06:54:15 (5.52 MB/s) - ‘k_means_model_data.pth’ saved [2106323/2106323]
```

[7]
```
optimized_model = torch.load('/content/optimized_model.pth')

# [k_loss, k_init_centroids, k_final_centroids, k_end_centroid_count] is the consturction of the optimized model
k_loss = optimized_model[0]
k_init_centroids = optimized_model[1]
k_final_centroids = optimized_model[2]
k_end_centroid_count = optimized_model[3]

# print(k_end_centroid_count)
print('loaded model successfully')

# now need to load the test set as a model
k_means_data = torch.load('/content/k_means_model_data.pth')

# [training_set, testing_set, LAT_RANGE, LNG_RANGE, LAT_SHIFT, LNG_SHIFT]
training_set = k_means_data[0]
testing_set = k_means_data[1]
LAT_RANGE = k_means_data[2]
LNG_RANGE = k_means_data[3]
LAT_SHIFT = k_means_data[4]
LNG_SHIFT = k_means_data[5]
# print("lat range")
# print(LAT_RANGE)
# print("lng range")
# print(LNG_RANGE)
# print("lat shift")
# print(LAT_SHIFT)
# print("lng shift")
# print(LNG_SHIFT)
# print('training set')
# print(training_set)
# print('testing set')
# print(testing_set)
# print(' ')
print('loaded k means data successfully')
```

```
loaded model successfully
loaded k means data successfully
<ipython-input-7-29822c86fe73>:1: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which
  optimized_model = torch.load('/content/optimized_model.pth')
<ipython-input-7-29822c86fe73>:13: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data whic
  k_means_data = torch.load('/content/k_means_model_data.pth')
```

[66]
```
# optimized_model = [k_loss, k_init_centroids, k_final_centroids, k_end_centroid_count]
ks = [i for i in range(350)]
centroid_reduction = [ks[i] - k_end_centroid_count[i] for i in range(len(ks))]
# print(len(ks), len(k_loss))

def draw_k(K = 225, Show_Initial_Centroids=False, Loss_Start=25):
  # Initialise the subplot function using number of rows and columns
  # figure, axis = plt.subplots(2,2, figsize=(30, 10))
  # draw_counties(), recreated for subplots:
  fig = plt.figure(figsize=(12, 8))
  gs = fig.add_gridspec(3,2)
  ax1 = fig.add_subplot(gs[0:2, :])
  ax2 = fig.add_subplot(gs[2, 0])
  ax3 = fig.add_subplot(gs[2, 1])

  ax1.scatter(draw_lnglat_tensor[:, 0], draw_lnglat_tensor[:, 1], c='#79d2a4', marker='.', s=4, label='U.S. Counties')
  ax1.set_xlabel('Longitude')
  ax1.set_ylabel('Latitude')
```

```
    if Show_Initial_Centroids:
      ax1.scatter(k_init_centroids[K][:, 0], k_init_centroids[K][:, 1], c='r', s=50, marker='1', label='Randomized Initialization')
    ax1.scatter(k_final_centroids[K][:, 0], k_final_centroids[K][:, 1], c='b', s=50, marker='2', label=f'{k_end_centroid_count[K]} Optimized Centroids')
    ax1.set_title(f'K-Means Optimized Centroids For K={K}')
    ax1.legend(loc='lower left')

    left_bound = 1 if not Loss_Start else Loss_Start
    ax2.plot(ks[left_bound:], k_loss[left_bound:], c='#56a0d3', marker='o', label='All Losses')
    ax2.plot([K], k_loss[K], c='r', marker=7, markersize=25, label=f'Loss At K={K}')
    ax2.set_title(f'For {K} Centroid{"s" if K > 1 else ""} Euclidian-Derived Loss Was {round(k_loss[K],3)}')
    handles, labels = ax2.get_legend_handles_labels()
    ax2.legend(reversed(handles), reversed(labels), loc='center right')

    # finally, just a simple plot of how many centroids were lost at each value of k
    ax3.bar(ks, centroid_reduction, color='#56a0d3', label='Removed Centroids')
    ax3.plot([K], centroid_reduction[K], c='r', marker=7, markersize=25, label='Starting K Centroids')
    ax3.set_title(f'For {K} Starting Centroid{"s" if K > 1 else ""}, {centroid_reduction[K]} {"Were" if centroid_reduction[K] != 1 else "Was"} Removed For Redundance')
    ax3.legend(loc='center left')

    # Instead of converting all the data back to normal latitudes and longitudes, I'm gonna just hack the axis x_labels
    # but for subplots need to finesse this with the .sca command
    plt.sca(ax1)
    x_locs, x_labels = plt.xticks()
    x_labels = [int(item + MEAN_LNG) for item in x_locs]
    plt.xticks(x_locs, x_labels)
    y_locs, y_labels = plt.yticks()
    y_labels = [int(item + MEAN_LAT) for item in y_locs]
    plt.yticks(y_locs, y_labels)

    plt.tight_layout()
    plt.show()
    return

# plot = interact(draw_k, K=(1, 349), Loss_Start=(0, 345, 25))
```

```
# matplotlib 3d plots don't work with colab notebooks so here's a plotly version
# produced some plotly code with Claude ai
import numpy as np
import plotly.graph_objs as go

# Plotly 3D Trisurf Plot with Custom Color Scale
color_options = {
    'Deep': 'red',
    'Temps': 'black',
    'Armyrose':'black',
    'Curl':'orange',
    'BrBG':'black',
    'Tealrose':'black'
}

def create_trisurf_plot(ColorScheme='Tealrose', Show_K_Centroids=False, K=225):
    """
    Create a 3D trisurf plot using Plotly with custom color scale

    Parameters:
    x (array-like): x-coordinates
    y (array-like): y-coordinates
    z (array-like): z-coordinates
    vmin (float, optional): Minimum value for color scaling
    vmax (float, optional): Maximum value for color scaling

    Returns:
    Plotly Figure object
    """
    y = drawing_df['lat'].to_list()
    x = drawing_df['lng'].to_list()
    z = drawing_df['population'].to_list()

    vmin = 5
    vmax=None
    opacity_choice = 0.575 if Show_K_Centroids else 1

    # If vmin or vmax are not provided, calculate them
    if vmin is None:
        vmin = np.percentile(z, 1)  # 1st percentile
    if vmax is None:
        vmax = np.percentile(z, 99)  # 99th percentile

    # Create the trisurf plot
    trace = go.Mesh3d(
        x=x,
        y=y,
        z=z,
        colorscale=ColorScheme,  # You can change this to other color scales
        intensity=z,
        cmin=vmin,  # Set minimum for color scaling
        cmax=vmax,  # Set maximum for color scaling
        showscale=True,
        opacity=opacity_choice
    )
    # Create a scatter plot of k values if desired
    # if turn_on_scatter:

    scatter_x = k_final_centroids[K][:, 0]
    scatter_y = k_final_centroids[K][:, 1]
    scatter_z = [-1 for i in range(K)]
    scatter_trace = go.Scatter3d(
        x=scatter_x, #k_final_centroids[225][:, 0],
        y=scatter_y, #k_final_centroids[225][:, 1],
        z=scatter_z, #[1000000 for i in range(225)],
        mode='markers',
        marker=dict(
            size=3,
            color=color_options[ColorScheme],
            opacity=1
        )
    )

    # calculate the ticks
    x_ticks = np.linspace(min(x), max(x), 5)  # 5 ticks across the range
    y_ticks = np.linspace(min(y), max(y), 5)
    z_ticks = np.linspace(min(z), max(z), 5)

    # Configure the layout
    layout = go.Layout(
        title=f'U.S. Population By County With {k_end_centroid_count[K]} Centroids' if Show_K_Centroids else 'U.S. Population By County',
        width=750,  # Width in pixels
        height=750,
        scene=dict(
            zaxis_title='Population',

            xaxis=dict(
                title='Longitude',
                tickvals=x_ticks,  # Original data points
                ticktext=[int(i + MEAN_LNG) for i in x_ticks],  # Custom tick labels
            ),
            yaxis=dict(
                title='Latitude',
                tickvals=y_ticks,  # Original data points
                ticktext=[int(i + MEAN_LAT) for i in y_ticks],  # Custom tick labels
            ),
        )
```

```
    )

    # Create the figure
    plot_data = [trace]
    if Show_K_Centroids:
        plot_data.append(scatter_trace)
    fig = go.Figure(data=plot_data, layout=layout)
    fig.update_layout(
        scene=dict(
            camera=dict(
                eye=dict(x=0.25, y=-1.25, z=1.15)  # Adjust viewing angle
            )
        )
    )
    fig.show()

    return



def draw_trisurf(ColorScheme='Tealrose', K=225, Show_K_Centroids=False):
    # Create and show the plot
    # fig = go.Figure()
    fig = create_trisurf_plot(ColorScheme, K, Show_K_Centroids)
    fig.show()
    return fig

# fig = create_trisurf_plot(ColorScheme ='Tealrose', opacity_choice=0.7, Show_K_Centroids=True, K=225)
# plot = interact(create_trisurf_plot, ColorScheme=list(color_options.keys()), K=(1, 349), opacity_choice=(0.1, 1.0, 0.1))
# If you want to save the plot
# fig.write_html("trisurf_plot.html")

# and finally create a dropdown interaction bar with all the coolest looking colors
# https://plotly.com/python/builtin-colorscales/
```

[29]

```
# statistical analysis of data to test data
# loss function
def find_closest_centroids_and_loss(X, centroids):
    """ compute the centorid memberships for each example
    we need to return two tensors:
        one that will return the indices of the centroid closest to each example
        one that will contain all of the loss for each example (min_dist by euclidian distance)
    """
    # cdist(X, centroids) produces a tensor where each row represents an input X, and each y is the euclidian distance to the centroid there
    euclidian_dist_to_centroids = torch.cdist(X, centroids, p=2)
    # rather than using .min and then .argmin to get the min and argmin, you can use a dimension argument in min to get both at the same time
    min_dist, argmin_indices = torch.min(euclidian_dist_to_centroids, dim=1)
    return (argmin_indices, min_dist)

def compare_averages(group1, group2, alpha=0.05):
    """
    Perform an independent t-test to compare two groups of data.

    Parameters:
    group1 (array-like): First group of numerical data
    group2 (array-like): Second group of numerical data
    alpha (float): Significance level for the test (default 0.05)

    Returns:
    dict: A dictionary containing test results
    """
    # Perform the t-test
    t_statistic, p_value = stats.ttest_ind(group1, group2)

    # Determine statistical significance
    is_significant = p_value < alpha

    # Calculate means and standard deviations
    mean1, std1 = np.mean(group1), np.std(group1)
    mean2, std2 = np.mean(group2), np.std(group2)

    return {
        't_statistic': t_statistic,
        'p_value': p_value,
        'mean1': mean1,
        'mean2': mean2,
        'std1': std1,
        'std2': std2,
        'is_significant': is_significant
    }

# Example usage
group1 = [1, 2, 3, 4, 5]
group2 = [2, 4, 6, 8, 10]

results = compare_averages(group1, group2)
print(results)

model_loss = [i / 100000 for i in k_loss[1:] ]
print(model_loss)
test_loss = []
# define a function to grab the test loss for any number of centroids
def test_k_loss(k):
    indices, losses = find_closest_centroids_and_loss(testing_set, k_final_centroids[k])
    loss_test_sum = losses.sum().item()
    avg_loss_test_sum = loss_test_sum / testing_set.shape[0]
    test_loss.append(avg_loss_test_sum)
# iterate through all of our model
for i in range(len(model_loss)):
    test_k_loss(i + 1)
print(test_loss)
# double check that our parameters work
print(len(model_loss) == len(test_loss))
# and finally perform ttest on the two vectors
compare_averages(model_loss, test_loss)

```

```
{'t_statistic': -1.8973665961010275, 'p_value': 0.09434977284243756, 'mean1': 3.0, 'mean2': 6.0, 'std1': 1.4142135623730951, 'std2': 2.8284271247461903, 'is_significant': False}
[14.349576309487706, 8.100688949003201, 6.218209798876435, 5.002503813215761, 5.002503813215761, 4.781395406030207 5, 3.876655910648112, 3.4733336817257436, 3.2142986949018244, 3.0052037897259085, 2.9308566005540175, 2.7885587592561065,
[14.436541348750596, 8.108987117191914, 6.2244319588496415, 5.006456319244491, 5.006456319244491, 4.782516425783155, 3.8780205271812536, 3.476943785336638, 3.213319359819653, 3.011018457309649, 2.926733952989744, 2.777073705146354, 2.5
True
{'t_statistic': 0.048656859589206986,
 'p_value': 0.9612067150332868,
 'mean1': 0.86819473414393,
 'mean2': 0.8642205618811357,
 'std1': 1.0755877882149576,
 'std2': 1.0792104842697057,
 'is_significant': False}
```

```
significant_k_values = []
for i in range(len(test_loss)):

    cur_ttest = compare_averages([model_loss[i]], [test_loss[i]])
    # print(f'hi {i}, {cur_ttest["is_significant"]}')
    if cur_ttest["is_significant"]:
        significant_k_values.append(i)

if not significant_k_values:
    print('all k values received False on their ttest comparison')
else:
    print(significant_k_values)
```

```
/usr/local/lib/python3.10/dist-packages/scipy/stats/_stats_py.py:6951: RuntimeWarning:

invalid value encountered in scalar divide

all k values received False on their ttest comparison
```

Before presenting the results, here is a brief explanation to the work

You can see the full process on my original colab, https://colab.research.google.com/drive/1HOYoMWFS5AqRRV1Ay944549crisrOtmK?usp=sharing

As well as the followup https://colab.research.google.com/drive/1PG4R404VJgmgf_hGXeHV6vrC5EsE3Kxr?usp=sharing that was mostly testing that the export and import of the pytorch worked correctly before creating this results page.

What was the point of this project?

There are several countries in the world that have developed High Speed Railway networks across their nations. In recent years I have seen proposed railway networks in the U.S., once in California and once on the East Coast.

I believe that while we argue over EVs, the real issue is that travel in the U.S. has become too car-dependant, and that an efficient railway network across the country could revolutionize travel and commerce.

When I took Andrew NG's Machine Learning Specialization courses I wrote down a lot of ideas for projects for each of the topics. One I had while watching the section on Unsupervised Learning, was the machine learning algorithm K-Means to cluster data points.

I thought that by taking the entirety of the U.S. Population, you could use K-Means clustering to determine the most efficient number and placement of railway stations to best serve the U.S. population.

-- K-Means would only create the points, but not the connections. For further study I hope one day to create some kind of algorithm that would reproduce this slime mold experiment to connect the points. https://phys.org/news/2022-01-virtual-slime-mold-subway-network.html A traveling salesman problem would be an initial start, but I would want to somehow weigh the geography and barriers to building railway networks in certain parts of the country, so this is outside of the scope of the experiment.

The Inputs

I originally looked to get my data from census.gov, but that would have involved downloading the data for each state and putting it together myself. So I did a bit of searching and found this website that has county data for the United States in csv form, updated to May 26, 2024.

https://simplemaps.com/data/us-counties

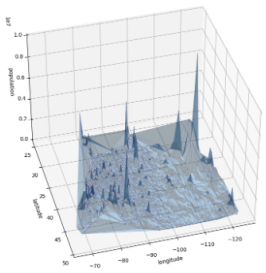
That data was perfect for this study. CSV is easy to work with, and the data was almost exactly in the form that I wanted.

All I had to do was use Pandas to parse it, and strip away any of the counties from Alaska and Hawaii so that we could only consider the contiguous U.S., and flatten the data.

Rather than using the data with 3 dimensions (lat, lng, and pop), I felt that if you broke down each county's population into data points representing x people, then it would be easier to implement a ML model to analyze the data.

These steps are on another github repository here:

https://github.com/Tyrowo/us-county-data-processing-and-visualization



Trisurf plot of the U.S. County Population data shown from Canada's perspective

The Algorithm

A K-Means algorithm is a pretty simple process:

1. You choose an arbitrary number of "Centroids," K, and randomize initial start points for them. A centroid represents the central location of a group of data points surrounding it.
2. Assign each point of data to its closest centroid.
3. For each centroid, take the location of all of the data points assigned to it and average them to find a new position representing the group's new center. If the centroid has no data points assigned to it it is discarded due to its redundance.
4. Repeat steps 2-3 until the centroids have found their final positions.

This process calculates the Loss of the model as the Euclidian distance for each point to its closest centroid. A better fitting model will have centroids that better represent the data by having centroids closer to the data.

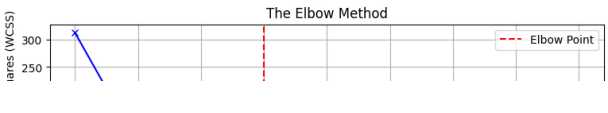
A standard optimization to this algorithm is to test many different initial randomized starting points for the centroids, and determining which of those creates the lowest ending loss for the model. I.e. depending on your starting points you can reach different end results, and those end results may be suboptimal.

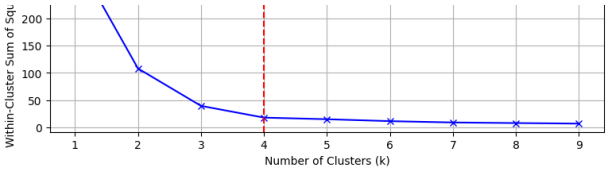
The Elbow

If you thought that then determining the "correct" number for K would then just be a matter of plugging in some formula, it is a bit more nuanced than you'd expect.

In K-Means it holds true that "more is better," in the sense that if you continue to add more centroids there will be lower loss to the model. Up until the point where you have the same amount of centroids as data points, and then any data point will just be assigned a centroid and extra centroids will be discarded.

So there exists an "elbow point" so to speak. A value of K where you can observe that adding more centroids stops becoming as efficient, because the loss is not reduced as much as before.





GeeksForGeeks Elbow Method https://www.geeksforgeeks.org/elbow-method-for-optimal-value-of-k-in-kmeans/
– As a note I think this figure dramaticises the dropoff after the elbow point a little bit.

My Process

With the above knowledge of K-Means in mind, my brute force implementation was such that I would take 100 randomized seeds, run the K-Means algorithm each time, determine which seed performed best, and repeat for every possible value of K between 1 and the number of data points, which was about 3100 (representing the number of counties in the dataset).

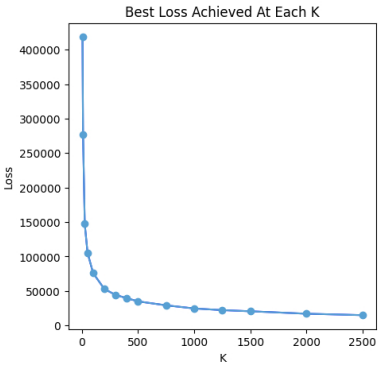
However, this would take WAY too long. Even after applying as much matrix multiplication as I could in determining which points should be assigned to which centroid, there is an issue with the step of reassigning the centroid locations in that it cannot be done with matrix multiplication (source: checked many websites) and even if you did it would be difficult to handle centroids that no longer had any data points assigned to them.

Thinking that I would be able to parallelize all of my operations with Cuda, initially my dataset was too large (1,000,000 data points representing 250 people per data point). To give you an impression of why that was too large, after reducing to 100,000 data points it took about an hour to perform the K-Means operation 100 times for each seed. For 3000 iterations of K that would be 3000 hours, or 125 days. So 1,000,000 was way too many.

So I determined two major goals for the project:

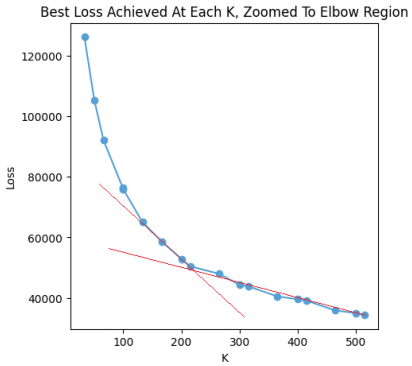
1. Figure out my elbow point early to know which values of K were most important to assess
2. Use "seeded" randomization built into pytorch to remember which seeds were producing the best results, and only use the optimal seed for the majority of the data.

I achieved this by taking a limited run of some values of K - 5, 10, 50, 100, 200, 300, 400, 500, 750, 1000, 1250, 1500, 2000, and 2500, and running them through 100 seeds to see what the loss values generally looked like.



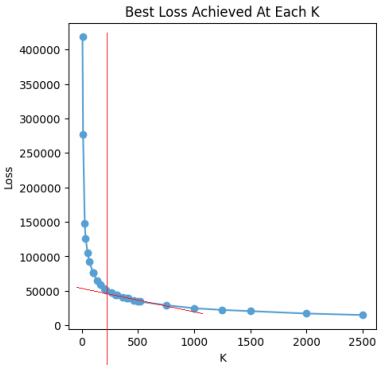
This was enough data to show pretty concretely that I could ignore anything higher than 500 - The elbow point appeared to be about 250.

So I did further testing, testing values of K primarily between 100 and 500. I reused the results from the previous test and did some slightly different values, testing K = 33, 66, 99, 133, 166, 215, 265, 315, 365, 415, 465, 515



From this graph I assessed that the elbow point was actually just past 215, where the steepness of the curve dropped off dramatically, flattening to a linear reduction in loss. For that reason I chose the elbow point to be 225.

Combining everything together it looks like this:



So you will notice in my resulting figures below I default the views to 225 because as it is our elbow point it is our accepted optimal K value.

From there I wanted to do some more work testing seeds. For seeds 1-100 there were two "optimal seeds" for values of K from 1 to 100 seed 34 was performing best, while every K value from 133-365 had 71 performing best. This was great news for me, because the seeds were performing consistently across the most important range of K values. It would not be necessary to test hundreds or even multiple seeds for each k, instead we could just run one optimal seed across every value.

So I did one final test of seeds from 100-251, testing 150 more seeds to compare to optimal seeds from the original 100. I used K=69 to represent the values from 1-110 and K=225 to represent values 111-350. And after a few hours of testing those 151 seeds on those two K values, it was determined that seed 102 performed best for K values of 1-110, and seed 71 was still the best for K values of 111-350.

When you click "Show Initial Centroids" you will see the centroids created from those two seeds.

Statistics and Generalization

Machine learning models are always trained on a specific set of data, and are trained by minimizing the loss of their models. But in their attempt to minimize loss, it is possible that the model can become overtuned to match the data it was trained on, minimizing their losses only on that Specific dataset. If you have a trained model and input fresh data, it should come to similar conclusions with the new data. If it comes to much less accurate conclusions with the new data than the training data, then the model does not "generalize" well, or it is "overfitted."

To test the generalization of my model I saved ~25% of the data as a test - 75% was to train the model and 25% was reserved to test when it was finished to test how well it generalizes.

After finding the final locations of the centroids and my optimal K value, I plugged in the test data and calculated the loss, finding the average loss over the sample size to compare how it performed against the training data.

The two values were very close (0.49755806230848587 vs 0.4930245952671911)

But I wasn't satisfied just eyeballing the loss values when I was comparing between the test data and the model data. So I found an easy way to implement a t-test with scipy, which is a statistical test to compare the significant difference between two means. I first ran the test on our optimal K value of 225 and found that there was not a statistically significant difference.

{'t_statistic': nan, 'p_value': nan, 'mean1': 0.4975580623084858 7, 'mean2': 0.4930245952671911, 'std1': 0.0, 'std2': 0.0, 'is_significant': False}

I then tested the averages for the the full list of average losses for each k, and found again that there was not significance in the difference of the means:

{'t_statistic': 0.048656860571886396, 'p_value': 0.9612067142504322, 'mean1': 0.86819473414393, 'mean2': 0.8642205618008149, 'std1': 1.0755877882149576, 'std2': 1.0792104843011172, 'is_significant': False}

Above you can see that I ran the test for each value of K as well, and found that none had a significant statistical difference.

With that, I felt that I had created an effective machine learning model!

Hopefully if you've made it here you now have a better understanding of what I did you can appreciate this data a little more.

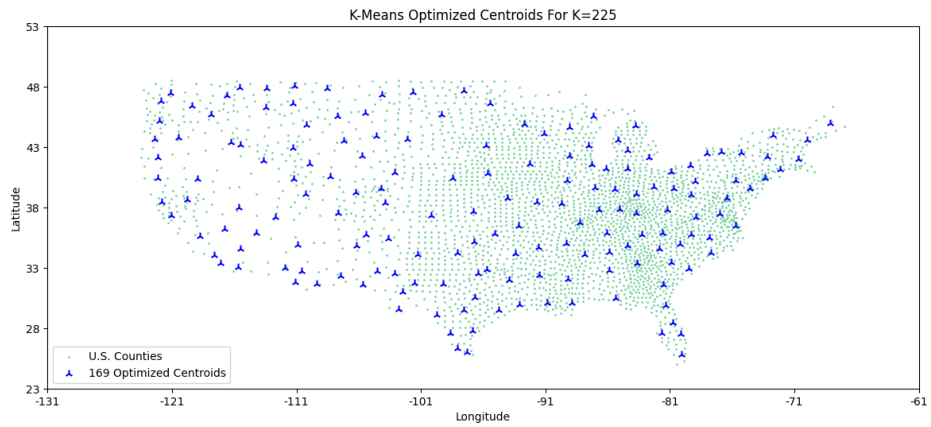
Here are final visualizations of the results

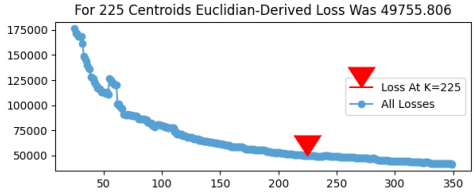
```
1 plot = interact(draw_k, K=(1, 349), Loss_Start=(0, 345, 25))
```

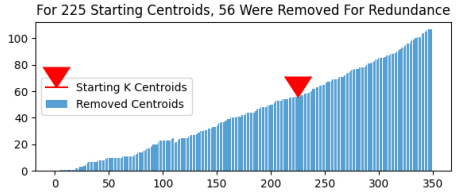
K 225

Show_Initial_Centroids

Loss_Start 25





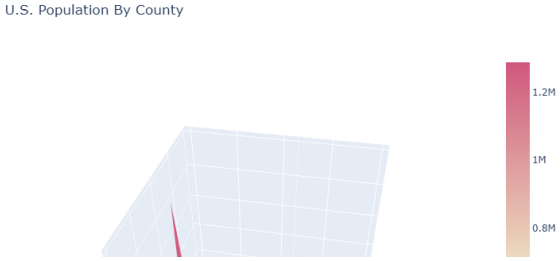


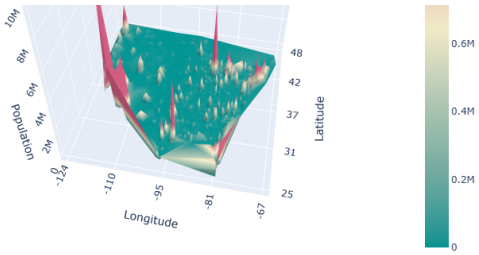

```
[32] 1 plot = interact(create_trisurf_plot, ColorScheme=list(color_options.keys()), K=(1, 349), opacity_choice=(0.1, 1.0, 0.1))
```

ColorScheme Tealrose

Show_K_Centroids

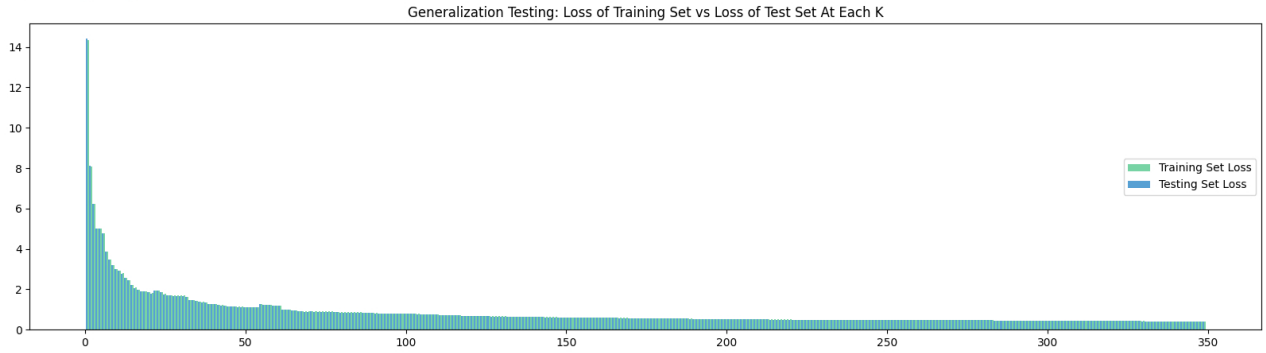
K 225





[84]
```
fig = plt.figure(figsize=(20, 5))
ks_half = [x-0.5 for x in ks]
plt.bar(ks[1:], model_loss, width=0.5, color='#79d2a4', label='Training Set Loss')
plt.bar(ks_half[1:], test_loss, width=0.5, color='#56a0d3', label='Testing Set Loss')
# fig.plot([K], centroid_reduction[K], c='r', marker=7, markersize=25, label='Starting K Centroids')
plt.title(f'Generalization Testing: Loss of Training Set vs Loss of Test Set At Each K')
plt.legend(loc='center right')
```

```
<matplotlib.legend.Legend at 0x7a0b382b3f70>
```



Takeaways and Improvements

1. I was really determined to use as much data as possible to prevent rounding errors, but some sacrifices need to be made. That being said, I think that 100,000 data points may have been a little bit of overkill.
2. I feel pretty limited with how little cuda I was able to apply. Other than pytorch built in methods I felt like I wasn't really taking advantage because I had to hardcode so much of my calculations
3. My elbow analysis was a little biased - performing Silhouette analysis may have been necessary to mathematically calculate the exact optimal point of k instead of using the elbow method. Maybe my rudimentary understanding of k-means led me to believe that 225 was optimal when it should have been much larger, when the line flattened again at 500 or again at 750. In reflecting I see that my inclination was biased towards choosing a smaller number to save time or to choose the fewest number of effective locations for the train stations. And although I only took a superficial look at Silhouette analysis, I believe that to calculate the distance from every point would have required that I generated the loss for many more values of K, all the way up to 3000. So while it would have created a more accurate depiction of which k was the best, it would not have been useful in saving time. Maybe if I had created a linear distribution of K through the dataset and performed silhouette analysis on that it may have worked.
4. The Los Angeles Problem - LA County produced a unique problem in that its population is over twice the amount of the second most populous county. As you can see from the map it is orders of magnitude larger than most counties in the United States. One thing I was thinking about while I was conceptualizing was that if you have so many clusters, that LA County ought to have more than one. Unfortunately that's not how k-means works. Due to all of the data points being on top of each other that was impossible, as any nearby clusters would be rendered redundant by the sole "LA Cluster." It would have been prudent to find some data specific to LA county and break it down further, so that all of those data points weren't sharing just one station. But that would require much more data, because LA County does not have any "subcounties" and would require breaking down the data in another way - instead of by using counties at all the experiment should probably done by city/town population instead of by county. Ultimately I decided to still assess the population by county for simplicity's sake.

Thanks for viewing my project!

If you have any questions feel free to reach out at tylerpcrews@gmail.com

```
-Tyler "Tyro" Crews
```