NYU CS-GY 6923: Machine Learning Fall 2024

Homework 4

Name: Rohan Mahesh Gore (N19332535)

Problem 1: Convolution for Shifted Images

Collaborators: None

We aim to design a convolutional feature transformation $C(I)$ that assists in correct classification of given images. The transformation is defined as:

$$C(I) = g(I \circledast W_1) \circledast W_2$$

where:
- W_1 and W_2 are 2×2 convolutional filters,
- $g(x)$ is an entrywise non-linearity.

Choice of Parameters:

1. Filters:

$$W_1 = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}, W_2 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

- W_1 captures diagonal patterns in the input image, which helps distinguish between shifted versions of the digits/
- W_2 aggregates extracted features to enhance classification.

2. For introducing non-linearity we use **Mish activation function**:

$$g(x) = x \cdot \tanh(\text{softplus}(x)) = x \cdot \tanh(\ln(1 + e^x))$$

- Mish is a self-regularizing non-monotonic activation that combines properties of ReLU and softplus.
- Mish is chosen because it is smooth, non-monotonic, and retains small negative values, which helps capture complex patterns in the data.

Python Code Implementation

```
import numpy as np

def softplus(x):
    return np.log1p(np.exp(x))

def mish(x):
    return x * np.tanh(softplus(x))

def convolve2d(image, kernel):
    output_height = image.shape[0] - kernel.shape[0] + 1
    output_width = image.shape[1] - kernel.shape[1] + 1
    output = np.zeros((output_height, output_width))

    for i in range(output_height):
        for j in range(output_width):
            output[i, j] = np.sum(image[i:i + kernel.shape[0], j:j + kernel.shape[1]] * kernel)

    return output

def C(I):
    W1 = np.array([[1, -1],
                   [-1, 1]])
    W2 = np.array([[1, 1],
                   [1, 1]])

    conv1 = convolve2d(I, W1)
    activated = mish(conv1)             # Apply Mish activation function
    conv2 = convolve2d(activated, W2)

    return conv2.flatten()

# images I1 to I8 given as input separately, in the form of ndarray with shape (5, 5)

C_I_One = C(I1)
C_I_Two = C(I2)

test_images = [I3, I4, I5, I6, I7, I8]

for i, Ii in enumerate(test_images):
    C_Ii = C(Ii)
    dist_to_I_One = np.linalg.norm(C_Ii - C_I_One)
    dist_to_I_Two = np.linalg.norm(C_Ii - C_I_Two)

    classified_class = 0 if dist_to_I_One < dist_to_I_Two else 1

    print(f"I{i + 3}: Dist_I1 = {dist_to_I_One:.4f}, Dist_I2 = {dist_to_I_Two:.4f} --> Classified as {c
```

Output

```
I3: Dist_I1 = 3.5250, Dist_I2 = 4.2600 --> Classified as 0
I4: Dist_I1 = 5.1702, Dist_I2 = 3.3953 --> Classified as 1
I5: Dist_I1 = 4.3737, Dist_I2 = 3.6616 --> Classified as 1
I6: Dist_I1 = 4.7729, Dist_I2 = 5.5081 --> Classified as 0
I7: Dist_I1 = 5.5947, Dist_I2 = 3.6790 --> Classified as 1
I8: Dist_I1 = 3.5250, Dist_I2 = 4.2600 --> Classified as 0
```

After calculating and analyzing the distances of points I3,....,I8 from the train images I1 and I2, we can say that the final classifications are correct as it classifies I_3, I_6, and I_8 as zero "0" and I_4, I_5, and I_7 as one "1".

Hence, Solved.

Problem 2: The necessity of non-linearities in CNNs

Collaborators: None

Step 1: Defining Operation with First Convolution:
Let's start with the convolution $x * W_1$ which can me defined as,

$$y[i] = \sum_{j=1}^{\ell} W_1[j] \cdot x[i+j-1]$$

This produces the intermediate output y, which now has size $d - \ell + 1$.

Step 2: Defining OPeration with Second Convolution:

We can apply the second filter W_2 on the result y which was received from the first convolution. It is,

$$z[m] = \sum_{n=1}^{k} W_2[n] \cdot y[m+n-1]$$

As, y is itself a result of a convolution (from step 1), we can replace $y[m+n-1]$ using its definition:

$$y[m+n-1] = \sum_{j=1}^{\ell} W_1[j] \cdot x[(m+n-1)+j-1] = \sum_{j=1}^{\ell} W_1[j] \cdot x[m+n+j-3]$$

Step 3: Substituting the value for $y[m+n-1$ in $z[m]$:

Thus by combining both the convolutions we get,

$$z[m] = \sum_{n=1}^{k} W_2[n] \cdot \left(\sum_{j=1}^{\ell} W_1[j] \cdot x[m+n+j-3] \right)$$

Use the distributive property of summation:

$$z[m] = \sum_{n=1}^{k} \sum_{j=1}^{\ell} W_2[n] \cdot W_1[j] \cdot x[m+n+j-3]$$

Simplifying and **Reorganize the indices to combine the sums**:

$$z[m] = \sum_{p=1}^{\ell+k-1} \left(\sum_{j=1}^{\ell} W_1[j] \cdot W_2[p-j+1] \right) \cdot x[m+p-2]$$

Step 4: Identifying the New Filter $\bar{W}$:

The sum inside the parentheses is precisely the definition of a convolution between the two filters W_1 and W_2. Let's denote this as:

$$\bar{W}[p] = \sum_{j=1}^{\ell} W_1[j] \cdot W_2[p - j + 1]$$

This is the formula for convolving W_1 (size ℓ) with W_2 (size k). One important note is that **the size of $\bar{W}$ is** $\ell + k - 1$.

Hence the Final Form for $z[m]$ can now be written as a single convolution between x and the new filter $\bar{W}$ as:

$$z[m] = \sum_{p=1}^{\ell+k-1} \bar{W}[p] \cdot x[m + p - 2]$$

This is precisely the definition of $x * \bar{W}$ which proves that $(x * W_1) * W_2 = x * \bar{W}$.

Hence Proved.

Problem 3: Triangulating Points via Principal Component Analysis

Collaborators: None

a. Recover XX^T from the distance matrix D

Step 1: Write down the distance in terms of inner products:

$$D_{i,j} = \|x_i - x_j\|_2^2 = \|x_i\|_2^2 + \|x_j\|_2^2 - 2\langle x_i, x_j\rangle$$

Rearranging for $\langle x_i, x_j\rangle$:

$$\langle x_i, x_j\rangle = \frac{1}{2}\left(\|x_i\|_2^2 + \|x_j\|_2^2 - D_{i,j}\right)$$

We can notice that $\|x_i\|_2^2$ appears as a constant for each row/column.

Therefore, if we define $b_i = \|x_i\|_2^2$, then we can write:

$$\langle x_i, x_j\rangle = \frac{1}{2}(b_i + b_j - D_{i,j})$$

Step 2: Utilizing the extra information (hint) $x_1 = 0$:

WE have been given that: If $x_1 = 0$, then $\|x_1\|_2^2 = 0$.

Using this in the equation for $\langle x_1, x_j\rangle$, we get:

$$\langle x_1, x_j\rangle = \frac{1}{2}(0 + \|x_j\|_2^2 - D_{1,j}) \implies \|x_j\|_2^2 = D_{1,j}$$

This tells us that $\|x_j\|_2^2$ can be extracted directly from the first row of D as $D_{1,j}$.

Step 3: Recovering XX^T:

We know that $\langle x_i, x_j\rangle$ is the (i,j)-th entry of the matrix XX^T.

Therefore, by using the result from step 1, we now know that:

$$(XX^T)_{i,j} = \frac{1}{2}(D_{1,i} + D_{1,j} - D_{i,j})$$

This equation gives us the inner product between any two points x_i and x_j directly from the distance matrix D.

Therefore to recover XX^T from the distance matrix D, we will compute:

$$(XX^T)_{i,j} = \frac{1}{2}(D_{1,i} + D_{1,j} - D_{i,j})$$

Hence Solved.

b. Use SVD of XX^T to recover loading vectors $z_1, \cdots, z_n$

Step 1: Computing the standard SVD of XX^T:

$$XX^T = U\Sigma U^T$$

where,
- U is an $n \times n$ orthogonal matrix (its columns are eigenvectors of XX^T)
- Σ is an $n \times d$ diagonal matrix with singular values $\sigma_1, \sigma_2, \ldots, \sigma_d$ on its diagonal

Step 2: Extract the first k components:

- Take the first k columns of U, denoted as U_k.
- Take the top k eigenvalues from Σ, denoted as Σ_k.

Step 3: Computing the loading vectors z_i:

The loading vectors are given by:

$$z_i = U_k \Sigma_k^{1/2}$$

These vectors are in $\mathbb{R}^k$ and are the lower-dimensional representations of the original points $x_1, \cdots, x_n$.

Hence, Calculated.

c. Proving that distances are preserved if $k = d$

Step 1: Compute the distance between original points x_i and x_j: Computing Euclidean distance between x_i and x_j:

$$\|x_i - x_j\|_2$$

Simplifying,

$$\|x_i - x_j\|_2^2 = x_i \cdot x_i - 2\, x_i \cdot x_j + x_j \cdot x_j$$

NOTE that $x_i \cdot x_i$ is the squared norm $\|x_i\|_2^2$, so this becomes:

$$\|x_i - x_j\|_2^2 = \|x_i\|_2^2 + \|x_j\|_2^2 - 2\langle x_i, x_j\rangle$$

Step 2: Using SVD of X:

- We know that X can be decomposed as

$$X = U\Sigma V^T$$

- If we think of x_i as the i-th row of X, we get:

$$x_i = (\text{i-th row of } X) = (\text{i-th row of } U\Sigma V^T)$$

- Let's denote the i-th row of U as $U_{i,:}$. Then:

$$x_i = U_{i,:} \cdot \Sigma \cdot V^T$$

Step 3: Computing the distance between z_i and z_j and comparing with distance between x_i and x_j:

- To compute the loading vectors z_i, we project X onto its first k principal components.
- If $k = d$, then all the components are retained. So: $z_i = x_i$
- In this case, the low-dimensional representations z_i and the original points x_i are exactly the same.
- Since $z_i = x_i$ (because $k = d$), the distance between z_i and z_j is the same as the distance between x_i and x_j:

$$\|z_i - z_j\|_2 = \|x_i - x_j\|_2$$

- This means that the low-dimensional vectors $z_1, \cdots, z_n$ are identical to the original points $x_1, \cdots, x_n$.

Hence, the distances between them remain unchanged as no information is lost in the dimensionality reduction process.

d. Reconstructing US Cities Locations

We will be visualizaing the location of citites on a 2D plane. The data is loaded as ”datalines” which is a multi-line string. The reconstrcuted locations of the cities are plotted using altair.

Python Code Implementation:

```
import numpy as np
import altair as alt
import pandas as pd

def create_distance_matrix(data_lines):
    processed_lines = []
    for line in data_lines:
        parts = line.split()
        for i, part in enumerate(parts[1:], 1):
            try:
                float(part)
                processed_lines.append((' '.join(parts[:i]), parts[i:]))
                break
            except ValueError:
                continue

    n = len(processed_lines)
    D = np.zeros((n, n))
    cities = []

    for i, (city, distances) in enumerate(processed_lines):
        cities.append(city)
        for j, dist in enumerate(distances):
            D[i,j] = float(dist)**2

    return np.array(D), cities

def recover_gram_matrix(D):
    n = D.shape[0]
    J = np.eye(n) - np.ones((n,n))/n
    return -0.5 * J @ D @ J

def get_loading_vectors(gram_matrix, k=2):
    eigenvals, eigenvecs = np.linalg.eigh(gram_matrix)
    idx = eigenvals.argsort()[::-1]
    eigenvals = eigenvals[idx]
    eigenvecs = eigenvecs[:,idx]
    return eigenvecs[:,:k] @ np.diag(np.sqrt(np.maximum(eigenvals[:k], 0)))

# Process data
D, cities = create_distance_matrix(data_lines)
gram_matrix = recover_gram_matrix(D)
Z = get_loading_vectors(gram_matrix, k=2)

# Transform coordinates to match target orientation
Z_transformed = np.copy(Z)
```

```
Z_transformed[:, 0] = -Z_transformed[:, 0]
Z_transformed[:, 1] = -Z_transformed[:, 1]

# Create DataFrame for Altair
df = pd.DataFrame({
    'x': Z_transformed[:,0],
    'y': Z_transformed[:,1],
    'city': cities
})

# Create Altair chart
chart = alt.Chart(df).mark_circle(size=100).encode(
    x=alt.X('x:Q', title='Dimension 1'),
    y=alt.Y('y:Q', title='Dimension 2'),
    tooltip=['city']
).properties(
    width=800,
    height=600,
    title='Reconstructed US Cities Locations'
) + alt.Chart(df).mark_text(
    align='left',
    baseline='middle',
    dx=7
).encode(
    x='x:Q',
    y='y:Q',
    text='city'
)

# Adding grid lines
chart = chart.configure_axis(
    grid=True
).configure_view(
    strokeWidth=0
)

chart
```

Output Visualization:

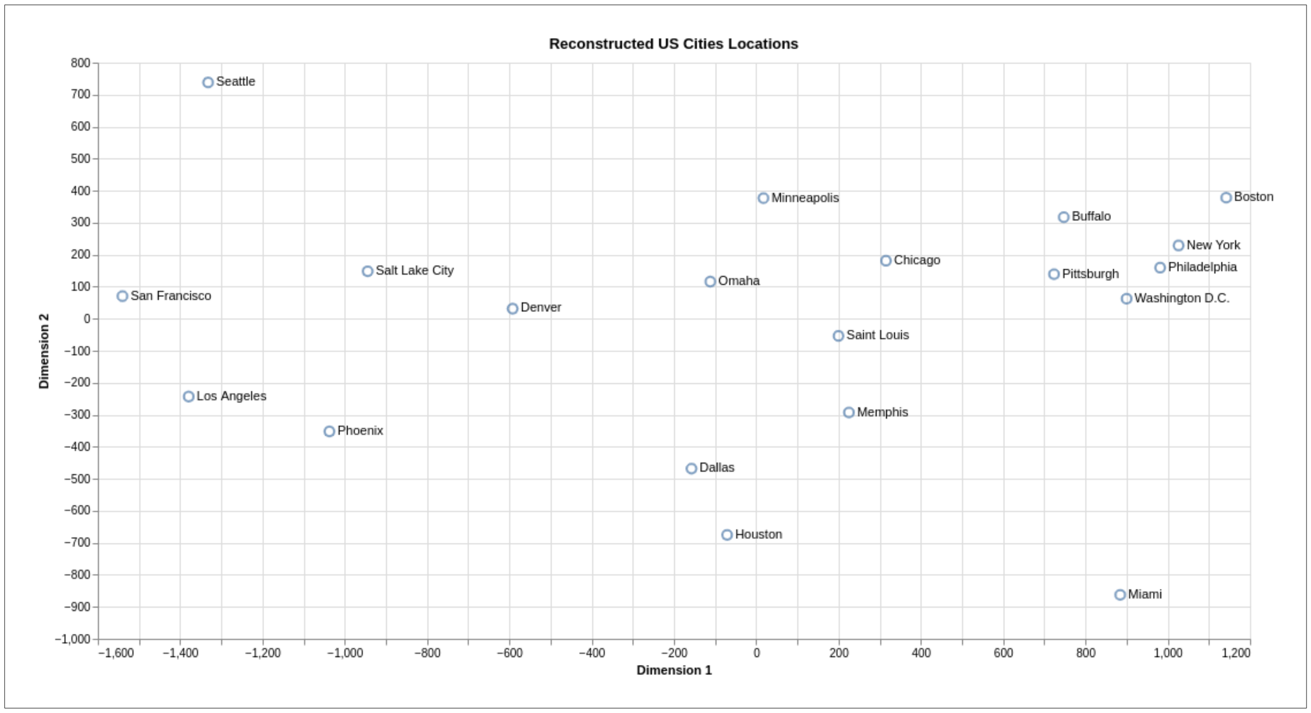

Figure 1: Reconstructed City Locations in 2D

e. Visualizations for Various Levels of Perturbation

Now we will be perturbing the distance and visualizing the same for different values of perturbation.

Python Code Implementation:

```
import numpy as np
import altair as alt
import pandas as pd

def create_distance_matrix(data_lines, r=0.0):
    processed_lines = []
    for line in data_lines:
        parts = line.split()
        for i, part in enumerate(parts[1:], 1):
            try:
                float(part)
                processed_lines.append((' '.join(parts[:i]), parts[i:]))
                break
            except ValueError:
                continue

    n = len(processed_lines)
    D = np.zeros((n, n))
    cities = []

    for i, (city, distances) in enumerate(processed_lines):
        cities.append(city)

        for j, dist in enumerate(distances):
            dist = float(dist)
            perturbed_dist = dist * (1 + r * (2 * np.random.random() - 1))  # Perturb Distance
            D[i, j] = perturbed_dist ** 2           # Squaring distances, as instructed
            D[j, i] = D[i, j]                       # Ensure symmetry

    return np.array(D), cities

def recover_gram_matrix(D):
    n = D.shape[0]
    row_mean = np.mean(D, axis=1)
    col_mean = np.mean(D, axis=0)
    total_mean = np.mean(D)
    G = -0.5 * (D - row_mean[:, np.newaxis] - col_mean[np.newaxis, :] + total_mean)
    return G

def get_loading_vectors(gram_matrix, k=2):
    U, S, Vt = np.linalg.svd(gram_matrix)
    Z = U[:, :k] @ np.diag(np.sqrt(S[:k]))
    return Z

# Visualize the 2D embeddings of the cities using Altair

def visualize_cities(Z, cities, title):
    df = pd.DataFrame({
        'x': Z[:, 0],
        'y': Z[:, 1],
        'city': cities
    })
    chart = alt.Chart(df).mark_point(size=100).encode(
        x=alt.X('x:Q', title='Dimension 1'),
        y=alt.Y('y:Q', title='Dimension 2'),
        tooltip=['city']
    ).properties(
        width=800,
        height=500,
        title=title
    ) + alt.Chart(df).mark_text(
        align='left',
        baseline='middle',
        dx=7
    ).encode(
        x='x:Q',
        y='y:Q',
        text='city'
    )
    chart = chart.configure_axis(
        grid=True
    ).configure_view(
        strokeWidth=0
    )

    return chart

# Main_exec

perturbation_factors = [0.1, 0.2, 0.3, 0.4, 0.5]

for r in perturbation_factors:
    # Step 1: Create distance matrix
    D, cities = create_distance_matrix(data_lines, r=r)

    # Step 2: Recover Gram matrix
    gram_matrix = recover_gram_matrix(D)

    # Step 3: Get the 2D positions
    Z = get_loading_vectors(gram_matrix, k=2)

    # Step 4: Flip the positions (to match the original logic)
    Z = -Z  # Flip the orientation of the layout

    # Step 5: Visualize the cities
    title = f"2D Representation of U.S. Cities (r = {r})"
    chart = visualize_cities(Z, cities, title)
    chart.display()
```

Output Visualizations:

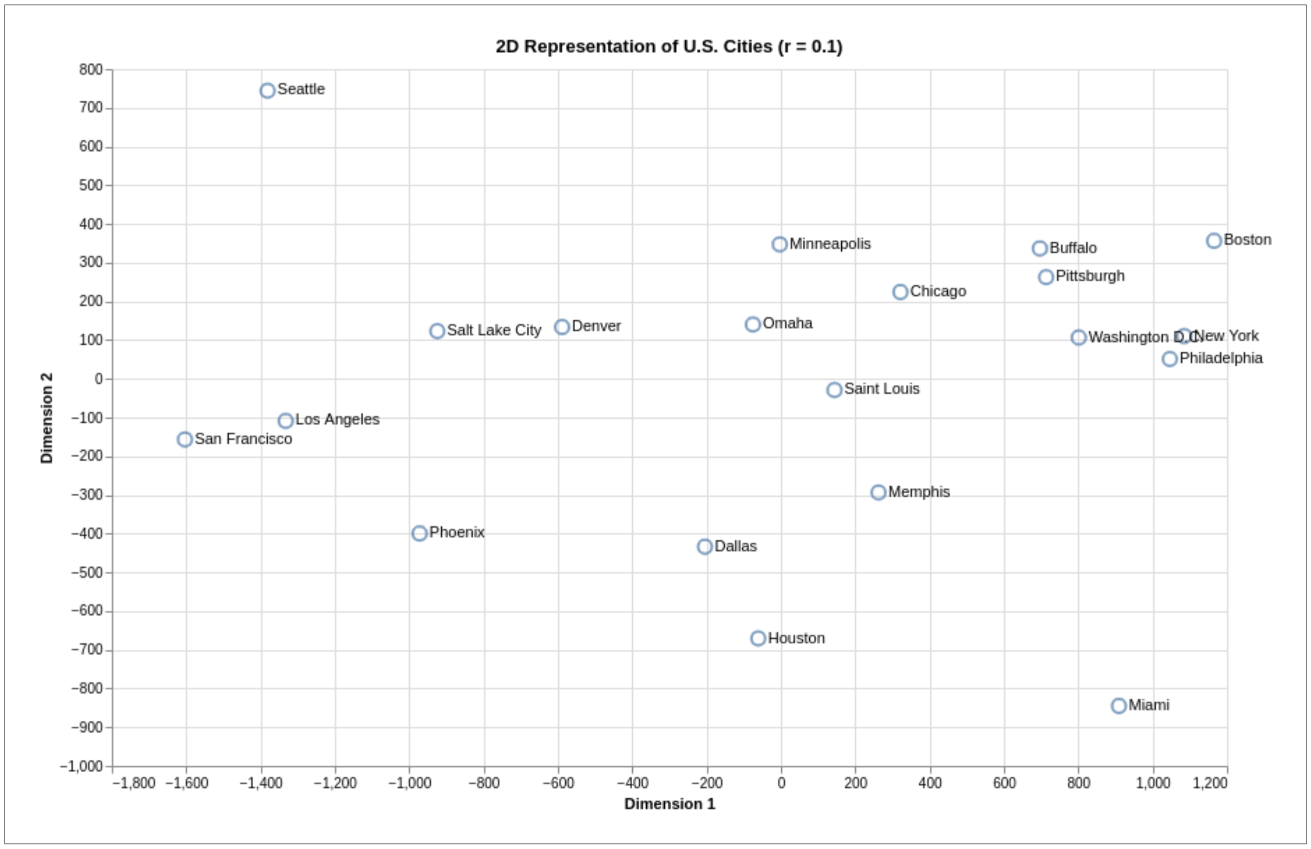

Figure 2: Reconstructed City Locations in 2D for r = 10%

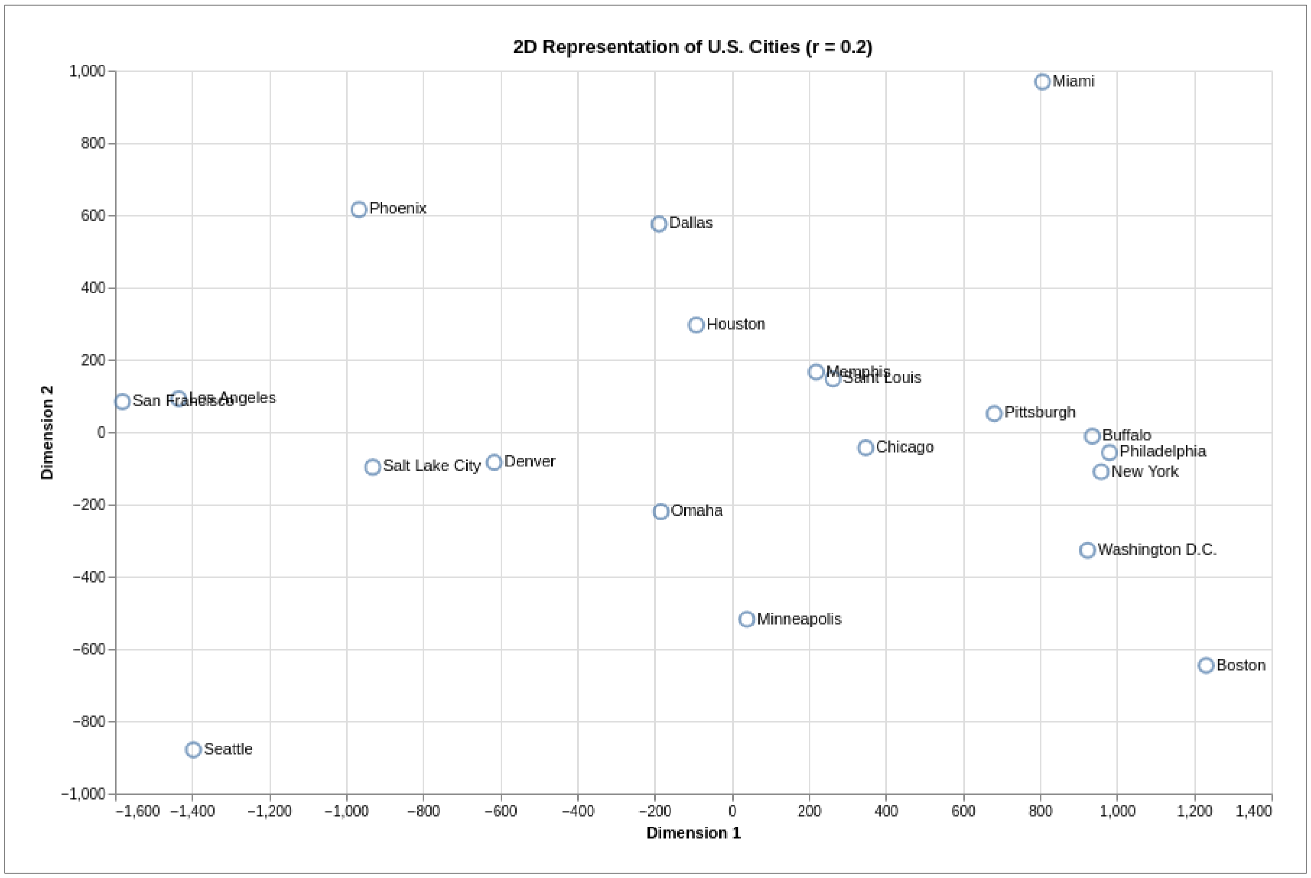

Figure 3: Reconstructed City Locations in 2D for r = 20%

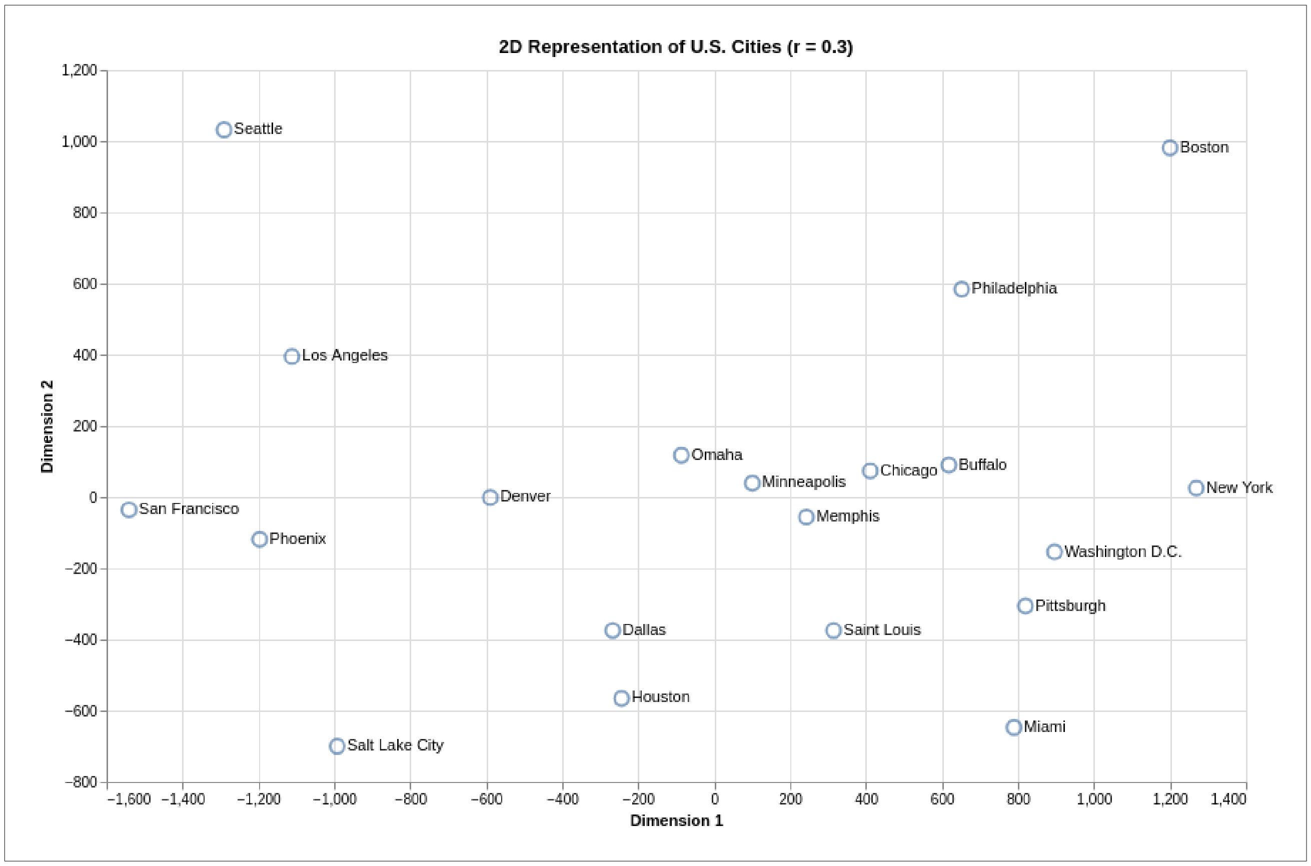

Figure 4: Reconstructed City Locations in 2D for r = 30%

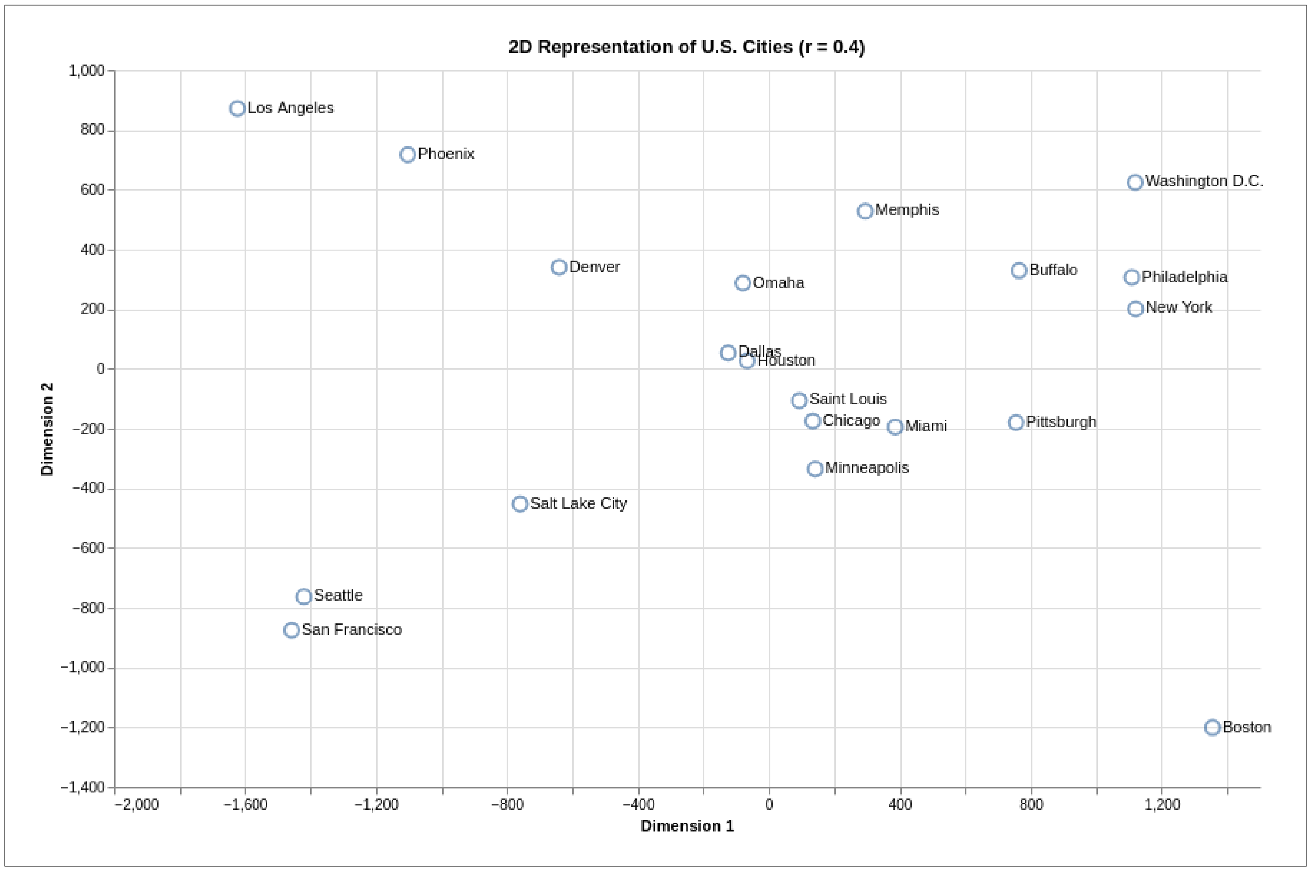

Figure 5: Reconstructed City Locations in 2D for r = 40%

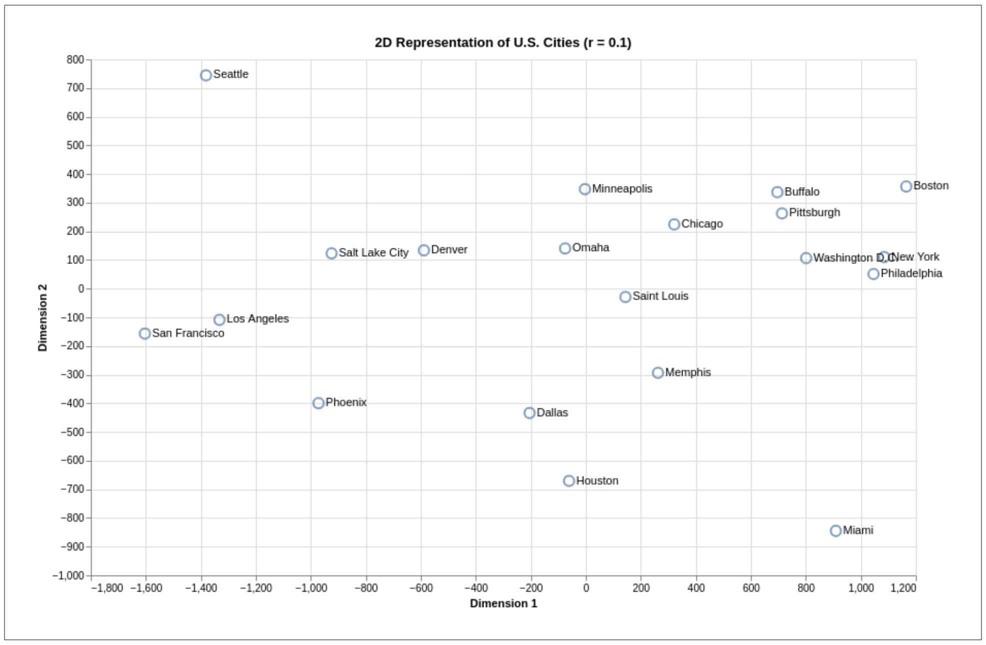

Figure 6: Reconstructed City Locations in 2D for r = 50%

- Using the above code we perturb the distances in the matrix and plot the new 2D representation for following levels of perturbation, that is where r = 0.1, 0.2, 0.3, 0.4 and 0.5.
- Inference for r = 0.1: The points are clearly maintaining the original structure of the point system and can be considered useful for any application.
- Inference for r = 0.2: There are a few major changes such as the complete mirroring of the location for Seattle and Miami. Also, location in the central area of the map, got cramped up due to slight changes in their positions. Looks like, the further away the points were from x-axis, more was their vertical displacement
- Inference for r = 0.3: There is an unusual of correctness which might be instantial, but nevertheless gives a good look at the locations of cities.
- Inference for r = 0.4, 0.5: At this point the locations completely lose the structural integrity and will not be helpful in anyway.
- Hence for a seting with "good approximation" and equivalent corretness of data points the perturbation value (r) should be around 20% or less.
- Beyond this threshold, the distortions become substantial enough to compromise the geographic integrity of the visualization, making it harder to reliably interpret the spatial relationships between cities.

Hence, the results demonstrate the robustness of MDS (Multidimensional Scaling) to perturbations (upto 20%) in the distance matrix.

Hence Demonstrated.