CA1:

Assignment Overview

You have **two main tasks**:

1. **Sampling with Langevin Dynamics**
2. **Tableau Dashboard and Storytelling**

https://colab.research.google.com/drive/1Yunh60WRexd04A7YetgXuFuzfnV3-zkW?usp=sharing

Task 1: Sampling

In this task, you will:

- Understand the **score function** and its relationship to probability distributions.
- Implement **Langevin Dynamics** to generate samples from a 2D Gaussian distribution.
- Compare the Langevin-generated samples with samples drawn directly using `numpy.random.multivariate_normal`.

1. Score Function

Consider a continuous probability distribution with probability density function p(x)p(\mathbf{x}). The **score function** is defined as:

∇xlogp(x).\nabla_{\mathbf{x}} \log p(\mathbf{x}).

Case: Unnormalized Probability Distribution

Often, we deal with probability distributions where the normalization constant is unknown or very difficult to compute. A general form for such a distribution can be written as:

p(x) = exp(fθ(x))Z(θ),p(\mathbf{x}) \;=\; \frac{ \exp\bigl(f_{\theta}(\mathbf{x})\bigr)}{Z(\theta)},

where:

- fθ(x)f_{\theta}(\mathbf{x}) is a function (often representing the negative energy or negative log-likelihood),
- Z(θ)Z(\theta) is the (unknown) normalization constant ensuring p(x)p(\mathbf{x}) integrates (or sums) to 1.

In such cases, the log-probability is:

$$\log p(\mathbf{x}) \;=\; f_\theta(\mathbf{x}) \;-\; \log Z(\theta).$$

When we differentiate with respect to x\mathbf{x}, the **score** function becomes:

$$\nabla_{\mathbf{x}} \log p(\mathbf{x}) \;=\; \nabla_{\mathbf{x}} \bigl(f_\theta(\mathbf{x}) - \log Z(\theta)\bigr) \;=\; \nabla_{\mathbf{x}} f_\theta(\mathbf{x}),$$

because $\log Z(\theta)$ is a constant with respect to x\mathbf{x} and its gradient is zero.

---

2. Langevin Dynamics

Langevin Dynamics is an algorithm to draw samples from a distribution when we only have access to the score function ∇xlogp(x)\nabla_{\mathbf{x}} \log p(\mathbf{x}). It is often formulated as follows:

1. **Initialize** a point X0\mathbf{X}_0 from some arbitrary distribution π\pi.
2. **Iterate** until convergence (or for a fixed number of steps TT):

$$\mathbf{X}_{t+1} \;\leftarrow\; \mathbf{X}_t \;+\; \epsilon\, \nabla_{\mathbf{x}} \log p(\mathbf{X}_t) \;+\; \sqrt{2\,\epsilon}\, \mathbf{Z}_t$$

where

- ϵ\epsilon is a **step size** (a small positive number),
- ∇xlogp(Xt)\nabla_{\mathbf{x}} \log p(\mathbf{X}_t) is the **score function** evaluated at Xt\mathbf{X}_t,
- Zt\mathbf{Z}_t is **Gaussian noise** sampled from a standard normal distribution (e.g. N(0,I)\mathcal{N}(\mathbf{0}, \mathbf{I})).

> Note: Some references write 2 ϵ Zt\sqrt{2\,\epsilon}\,\mathbf{Z}_t. Others may see a factor of σ\sigma for the noise term. The version given here matches a common form found in many machine learning contexts (especially in score-based generative modeling).

What You Need to Do (Step-by-Step)

1. **Define and Plot a 2D Gaussian Distribution**
   - Use a **2D Gaussian** with:
     $\boldsymbol{\mu} = \begin{bmatrix} -5 \\ 5 \end{bmatrix}, \quad \Sigma = 5\,\mathbf{I}.$
   - $\mathbf{I}$ is the 2×2 identity matrix, and $\Sigma = 5\mathbf{I}$ implies both variances are 5 and the correlation is 0.
   - Generate and plot the **contours** of this distribution in 2D space.
2. **Implement the Score Function**

   For a multivariate normal N(μ,Σ)\mathcal{N}(\boldsymbol{\mu}, \Sigma), the **log-density** (ignoring the normalization constant terms that do not depend on x\mathbf{x}) is:

   $f_\theta(\mathbf{x}) = -\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \Sigma^{-1}(\mathbf{x} - \boldsymbol{\mu}).$

   Hence, the **score** for a Gaussian distribution is:

   $\nabla_{\mathbf{x}} \log p(\mathbf{x}) \;=\; -\Sigma^{-1}(\mathbf{x} - \boldsymbol{\mu}).$
   - In your case,$\Sigma = 5\mathbf{I}$, so $\Sigma^{-1} = \frac{1}{5}\mathbf{I}$.
3. **Visualize the Score Function**
   - Create a grid of points in the 2D plane.
   - At each grid point, compute the vector

     $\nabla_{\mathbf{x}} \log p(\mathbf{x}) = -\frac{1}{5}(\mathbf{x} - \boldsymbol{\mu}).$
   - Use a **quiver plot** to show arrows pointing in the direction of the score at each grid point.
4. **Implement Langevin Dynamics**
   - **Input**: A set of initial points $\{\mathbf{X}_0^{(i)}\}$.
   - **Process**: For each point, update it using:

     $\mathbf{X}_{t+1}^{(i)} \;\leftarrow\; \mathbf{X}_t^{(i)} \;+\; \epsilon\,\nabla_{\mathbf{x}} \log p\big(\mathbf{X}_t^{(i)}\big) \;+\; \sqrt{2\epsilon}\,\mathbf{Z}_t^{(i)},$

     over T iterations.
   - **Output**: The final points $\{\mathbf{X}_T^{(i)}\}$ (these are your **Langevin samples**).
5. **Track and Visualize the Trajectories**
   - To see how each point moves over time, store the positions at each iteration.
   - Plot those trajectories on top of the distribution or in a separate figure.

6. **Compare Sampling Methods**
    - Draw **1,000 samples** using your **Langevin Dynamics** implementation.
    - Draw **1,000 samples** using `numpy.random.multivariate_normal(mean, cov, size=1000)`.
    - Compare the results (visually and/or statistically).
        - Visual comparison: overlay the two sets of points.
        - Statistical comparison: look at the mean, covariance of each set of samples, or use metrics like the **KL divergence** if you wish to go deeper.

Bonus Question (5%)

If you replace a single Gaussian with a **mixture of two Gaussians**:

$$p(\mathbf{x}) \;=\; \alpha\,\mathcal{N}(\mathbf{x}\,;\,\boldsymbol{\mu}_1,\Sigma_1) \;+\; (1-\alpha)\,\mathcal{N}(\mathbf{x}\,;\,\boldsymbol{\mu}_2,\Sigma_2),$$

where 0<α<10 < \alpha < 1, can you still get **proper samples** using Langevin Dynamics?

1. **Hint**: The **score function** now comes from the log of a sum of exponentials:

$$\log p(\mathbf{x}) = \log\Big(\alpha\,\mathcal{N}(\mathbf{x};\,\boldsymbol{\mu}_1,\Sigma_1) + (1-\alpha)\,\mathcal{N}(\mathbf{x};\,\boldsymbol{\mu}_2,\Sigma_2)\Big).$$

2. You need to compute:

$$\nabla_{\mathbf{x}} \log p(\mathbf{x}) \;=\; \frac{\alpha\,\nabla_{\mathbf{x}}\Big(\log\mathcal{N}(\mathbf{x};\boldsymbol{\mu}_1,\Sigma_1)\Big)\,\mathcal{N}(\mathbf{x};\boldsymbol{\mu}_1,\Sigma_1) \;+\; (1-\alpha)\,\nabla_{\mathbf{x}}\Big(\log\mathcal{N}(\mathbf{x};\boldsymbol{\mu}_2,\Sigma_2)\Big)\,\mathcal{N}(\mathbf{x};\boldsymbol{\mu}_2,\Sigma_2)}{\alpha\,\mathcal{N}(\mathbf{x};\boldsymbol{\mu}_1,\Sigma_1)+(1-\alpha)\,\mathcal{N}(\mathbf{x};\boldsymbol{\mu}_2,\Sigma_2)}.$$

3. **Answer**: Discuss whether Langevin Dynamics can correctly sample a mixture distribution. (Hint: It **can**, but the dynamics may get stuck in local modes or might need additional modifications like **tempered transitions** or multiple restarts to capture both modes thoroughly.)

Task 2: Tableau Dashboard

In the second task, you switch from sampling theory to **data visualization and storytelling** using **Tableau**.

Given two datasets:

1. **Airbnb_Listings.xls**
    - **Neighbourhood** – Specific area of the listing
    - **Price** – Cost per night
    - **Availability 365** – Number of days a listing is available per year
    - **Calculated Host Listings Count** – The total number of listings a host has
    - **Host Id** / **Host Name**
    - **Listing Id** / **Listing Name**
    - **Last Review** – Date of the most recent review
    - **Minimum Nights**
    - **Number Of Reviews**
    - **Reviews Per Month**
    - **Room Type** – e.g., Entire home/apt, Private room, Shared room
2. **Neighborhood_Locations.xlsx**
    - **Neighbourhood**
    - **Neighbourhood Group** (broader classification)

- **Latitude**
- **Longitude**

What to Do

1. **Data Preparation in Tableau**
   - Import both datasets.
   - Join them properly on the **Neighbourhood** field so each location has its **Longitude** and **Latitude**.
   - Handle any null values if needed.
2. **Create Multiple Dashboards**
   - Instead of only **one** dashboard, create **at least three** dashboards. Each dashboard should address a **different aspect** of the Airbnb data story.
   - Incorporate **interactive elements**: filters, parameters, or actions that let the user dynamically explore the data (e.g., filtering by neighborhood, price range, etc.).
3. **Use of KPIs & Storytelling**
   - Show **key performance indicators (KPIs)**, e.g., average price, total listings, average reviews per month, etc.
   - Organize your dashboards into a **story**:
     1. **Introduction**: What questions are you trying to answer or what trends are you looking for?
     2. **Findings** (across multiple dashboards):
        - Examples:
          - Seasonal or temporal trends in reviews
          - Geographic price distribution (map)
          - Room-type availability across neighborhoods
     3. **Conclusion**: Summarize main findings or interesting patterns.
4. **Design Principles**
   - Use **Gestalt Principles** (proximity, similarity, etc.) to keep visualizations clear.
   - Apply **preattentive attributes** (color, size, position) to emphasize important points.
5. **Submission**
   - **Upload** your dashboards to **Tableau Public**.
   - Provide a **PDF document** containing:
     - The **Tableau Public link** to your dashboards.
     - A short written **summary** of the story behind each dashboard.
     - **Screenshots** of your worksheets and dashboards.
     - An explanation of which **filters, parameters, and KPIs** you used.

Evaluation Criteria

1. **Data Integration** – Properly joining/preprocessing data.
2. **Dashboard Quality** – Interactivity, clarity, organization.
3. **Visualization Variety** – At least three dashboards, each with multiple relevant charts.

4. **Storytelling** – A cohesive, structured narrative connecting all dashboards.
5. **Use of Interactivity** – Meaningful filters, parameters, and actions.
6. **Design Principles** – Good use of Gestalt and preattentive attributes.
7. **Group Participation** – Everyone can explain and defend their part.

Task 1: Sampling:

we do it in google colab .
https://colab.research.google.com/drive/1Yunh60WRexd04A7YetgXuFuzfnV3-zkW?usp=sharing

this are the explanation of the code :

University of Tehran – Data Science Course

Computer Assignment 1: Sampling & Data Visualization

Instructors: Dr. Bahrak, Dr. Yaghoobzadeh

TAs: Mohammad Reza Alavi, Mohammad Kavian, Fatemeh Mohammadi

Group Members:

- **Mohammad Taha Majlesi** – *Student ID: 810101504*
- **Mohammad Hossein Mazhari** – *Student ID: 810101520*
- **Alireza Karimi** – *Student ID: 810101492*

defining the x and y values :

Task 1: Sampling

```
[1]  import numpy as np
     import matplotlib.pyplot as plt
     from scipy.stats import multivariate_normal, wasserstein_distance, ks_2samp
```

1. Imports **NumPy** for numerical computations and **Matplotlib** for plotting data and visualizations.
2. Imports `multivariate_normal` from **SciPy** to work with multivariate Gaussian distributions.
3. Imports `wasserstein_distance` to compute the Wasserstein (Earth Mover's) distance between two 1D distributions.
4. Imports `ks_2samp` to perform the **Kolmogorov–Smirnov test**, which compares two 1D samples statistically.

```
[7]  np.random.seed(33)
     mean = np.array([-5, 5])
     cov = 5 * np.eye(2)
```

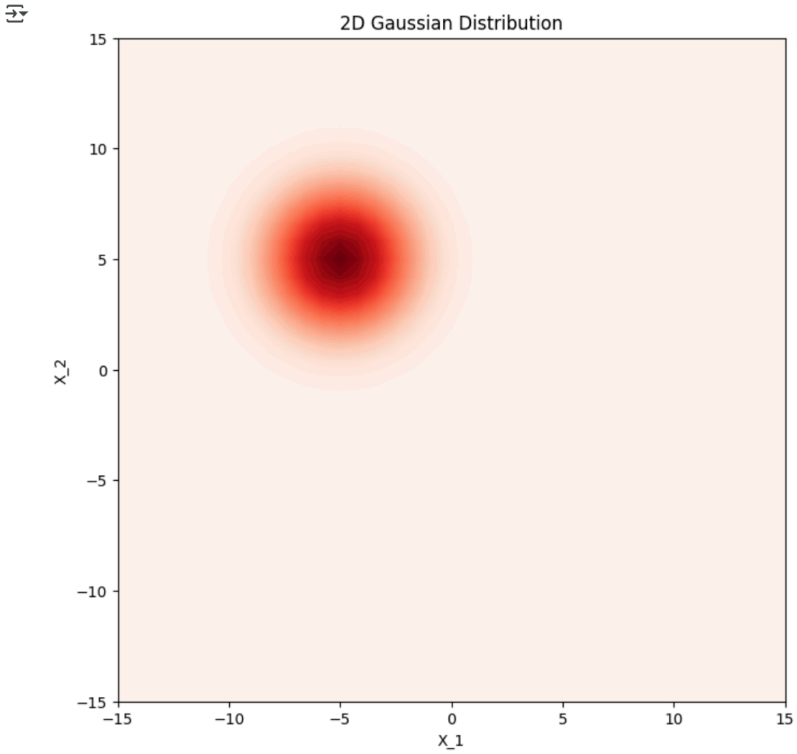
1. Sets a **random seed** (`33`) to ensure reproducible random results.
2. Defines the **mean vector** of the 2D Gaussian distribution as ([-5, 5]).
3. Creates a **2×2 identity matrix** using `np.eye(2)` and scales it by 5 to define the **covariance matrix**.
4. The resulting distribution is a **2D Gaussian** with independent variables and equal variance of 5 in each direction.

```
[8]  x = np.linspace(-15, 15, 40)
     y = np.linspace(-15, 15, 40)
     X, Y = np.meshgrid(x, y)
     pos = np.dstack((X, Y))
```

1. Creates 40 evenly spaced values from **-15 to 15** for both x and y axes using `np.linspace`.
2. Uses `np.meshgrid` to generate 2D coordinate matrices `X` and `Y` from the x and y arrays.
3. Stacks `X` and `Y` along the third axis using `np.dstack` to form a **(40×40×2)** array `pos`.
4. `pos[i, j]` now holds the 2D coordinate (x_i, y_j), used for evaluating the multivariate Gaussian PDF.

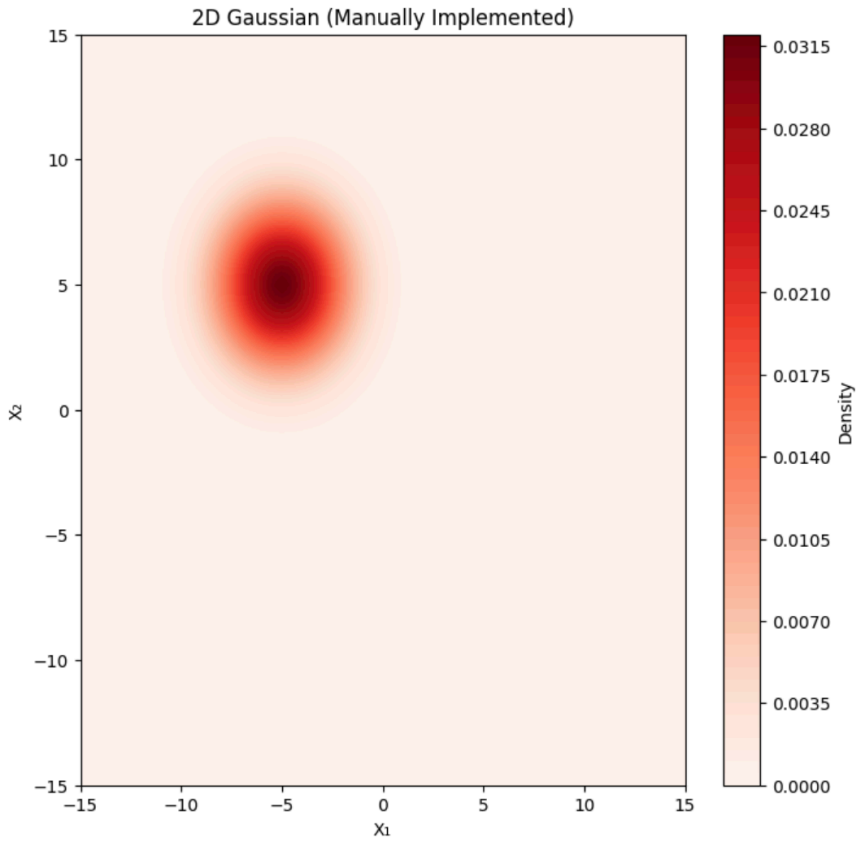
this is the plot of gaussian Distribution with mean of (5,5):

```
rv = multivariate_normal(mean, cov)
plt.figure(figsize=(8, 8))
plt.contourf(X, Y, rv.pdf(pos), cmap='Reds', levels=70)

plt.xlabel('X_1')
plt.ylabel('X_2')
plt.title('2D Gaussian Distribution')
plt.show()
```



We can also implement it manually in numpy

```
mean = np.array([-5, 5])
cov = 5 * np.eye(2)

x = np.linspace(-15, 15, 100)
y = np.linspace(-15, 15, 100)
X, Y = np.meshgrid(x, y)
pos = np.dstack((X, Y))

inv_cov = np.linalg.inv(cov)
det_cov = np.linalg.det(cov)

norm_const = 1.0 / (2 * np.pi * np.sqrt(det_cov))

diff = pos - mean
exponent = np.einsum('...i,ij,...j->...', diff, inv_cov, diff)
pdf_manual = norm_const * np.exp(-0.5 * exponent)

plt.figure(figsize=(8, 8))
plt.contourf(X, Y, pdf_manual, cmap='Reds', levels=70)
plt.xlabel('X₁')
plt.ylabel('X₂')
plt.title('2D Gaussian (Manually Implemented)')
plt.colorbar(label='Density')
plt.show()
```



Compute the Score Field (Vector Field) on the Grid:

Compute the Score Field (Vector Field) on the Grid

```
[15] U_field = np.zeros(X.shape)
     V_field = np.zeros(Y.shape)

     for row in range(X.shape[0]):
         for col in range(X.shape[1]):
             current_pt = np.array([X[row, col], Y[row, col]])
             grad_value = compute_score(current_pt, mean, cov_inverse)
             U_field[row, col] = grad_value[0]
             V_field[row, col] = grad_value[1]
```

```
[ ]  Start coding or generate with AI.
```

Score Field Computation

Initializes two matrices to store gradient components of the score function over a 2D grid. For each grid point ((x, y)), it computes the score vector:

$$\nabla_{\mathbf{x}} \log p(\mathbf{x}) = -\Sigma^{-1}(\mathbf{x} - \mu)$$

The x- and y-components of the gradient are stored in `U_field` and `V_field`, respectively, for later quiver plot visualization.

This code computes the gradient of the log-probability density function (i.e., the score function) at each point on a 2D grid by first initializing two arrays, `U_field` and `V_field`, with the same shape as the meshgrid arrays `X` and `Y` to store the x- and y-components respectively. Then, it iterates over every grid point using nested loops, constructs the current 2D point as a NumPy array, and computes its score via the function `compute_score` using the formula $\nabla_{\mathbf{x}} \log p(\mathbf{x}) = -\Sigma^{-1}(\mathbf{x} - \mu)$. The computed gradient components are saved into `U_field` and `V_field` for each grid coordinate, thereby constructing the full vector field that can be visualized using `plt.quiver(X, Y, U_field, V_field)`, which illustrates the direction of maximum increase in log-density over the 2D Gaussian distribution.

Define the Langevin Dynamics Function:

```python
def run_langevin(init_pts, score_fn=compute_score, num_steps=20, step=0.5):
    n_samples, n_dims = init_pts.shape
    trajectory = np.zeros((num_steps + 1, n_samples, n_dims))
    current_pts = init_pts.copy()

    trajectory[0] = current_pts.copy()

    for t in range(num_steps):
        scores = np.array([score_fn(current_pts[i], mean, cov_inverse) for i in range(n_samples)])
        noise_term = np.random.randn(n_samples, n_dims) * np.sqrt(2 * step)
        current_pts = current_pts + step * scores + noise_term
        trajectory[t + 1] = current_pts.copy()
    return current_pts, trajectory
```

This function implements **Langevin dynamics** to sample from a distribution using its **score function**. It iteratively updates each sample point $\mathbf{x}_t$ as:

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \epsilon \cdot \nabla_{\mathbf{x}} \log p(\mathbf{x}_t) + \sqrt{2\epsilon} \cdot \square(0, I)$$

Inputs:

- `init_pts`: Initial samples, shape (n, d).
- `score_fn`: Function to compute the score $\nabla_{\mathbf{x}} \log p(\mathbf{x})$.
- `num_steps`: Number of Langevin iterations.
- `step`: Step size parameter ϵ.

Outputs:

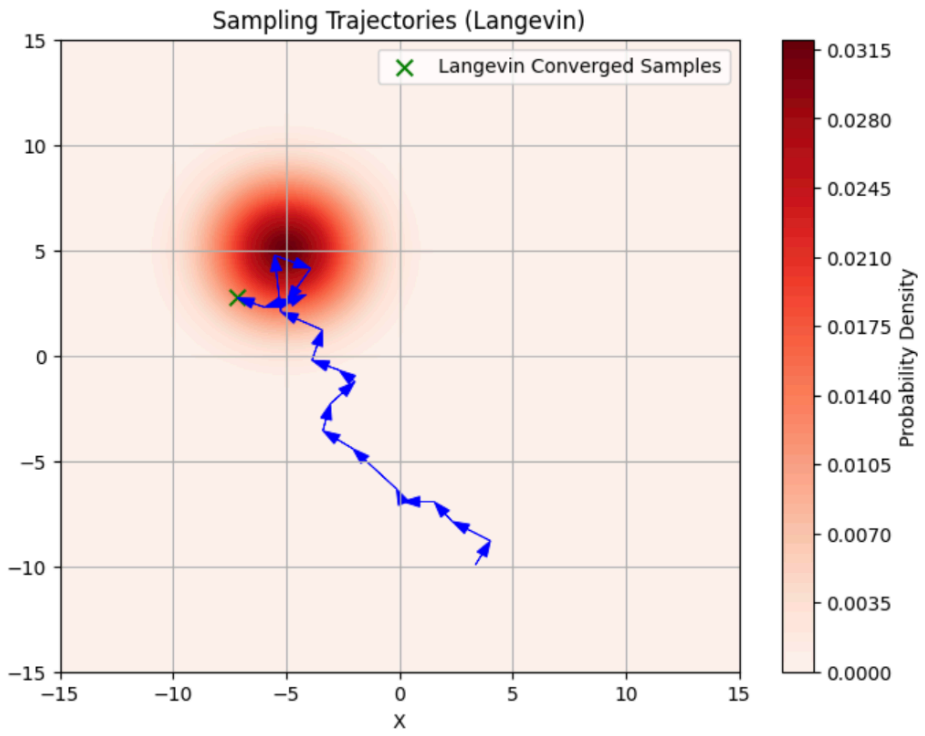
- `current_pts`: Final samples after T steps.
- `trajectory`: Full trajectory of samples over all iterations, useful for visualization.

Sampling Trajectories (Langevin):

```python
[19] np.random.seed(111)
     init_pts = np.random.uniform(low=[-15, -15], high=[15, 15], size=(1, 2))

     final_samples, traj = run_langevin(init_pts)
```

```python
plt.figure(figsize=(8, 6))

contour_filled = plt.contourf(X, Y, rv.pdf(pos), cmap='Reds', levels=70)

cbar = plt.colorbar(contour_filled)
cbar.set_label('Probability Density')
plt.scatter(final_samples[:, 0], final_samples[:, 1], c='green', marker='x', s=70, label='Langevin Converged Samples')

for i in range(len(traj) - 1):
    dx = traj[i + 1, 0, 0] - traj[i, 0, 0]
    dy = traj[i + 1, 0, 1] - traj[i, 0, 1]
    plt.arrow(
        traj[i, 0, 0], traj[i, 0, 1],
        dx, dy,
        shape='full', lw=0.6, length_includes_head=True, head_width=0.5, color='blue'
    )

plt.grid(True)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Sampling Trajectories (Langevin)')
plt.legend()
plt.show()
```

result :



The figure illustrates a **2D Gaussian distribution** (shown via a red-hued contour fill) overlaid with a **Langevin dynamics trajectory** (blue line) starting from an initial random point. Each segment of the trajectory reflects an update step according to the gradient of the log-density plus stochastic noise. The **green 'x'** marker denotes the final converged sample location, which lies in a high-density region of the Gaussian, as indicated by the dark red shading.

This visualization highlights the **iterative nature** of Langevin sampling, where the point steadily migrates from a low-density area toward the distribution's center of mass. The combination of **gradient-driven movement** and **random perturbations** helps the sampler overcome small local irregularities and converge near the mean. Consequently, this trajectory showcases how Langevin dynamics adapts both a **deterministic pull** (toward higher probability regions) and **stochastic exploration** (ensuring diversity in sampling paths).

(1) Background on the Plot

The figure depicts a **2D Gaussian distribution**—visualized as a heatmap (with higher intensity in the red region around the mean) and outlined by a corresponding color bar on the right. The mean of this Gaussian lies near the darkest red spot, indicating the area of highest probability density. A single **sampling trajectory** (blue line) is overlaid, showing how a point, initially placed in a low-density region, moves across the 2D plane as it undergoes Langevin updates.

(2) Core Idea of Langevin Dynamics

Langevin dynamics is a technique for sampling from a probability distribution when you have access only to its **score function** (gradient of log-density). At each step, the point is nudged by two factors:

1. A **deterministic drift**, which pushes it toward areas of higher density as determined by the gradient, and
2. A **stochastic term**, which adds random Gaussian noise to prevent the sampler from getting stuck in one exact location and to explore different regions of the distribution.

(3) Step-by-Step Trajectory

Looking at the blue path, the point starts relatively far from the Gaussian's mean. With each iteration, it takes a step in the **direction of the gradient**—pointing "downhill" in negative energy or "uphill" in log-density—plus a random jitter from the noise term. Because the gradient is stronger when the point is far from the mean, the trajectory initially makes fairly large jumps toward the high-density region.

(4) Convergence to High-Density Region

As the point nears the Gaussian's peak (where the red color is darkest), the gradient becomes subtler (less forceful) because it is closer to equilibrium. Consequently, updates become more finely balanced between local drift and noise. The **green '×'** marks the location where sampling (or the visualization) concludes, which is near the mean of the distribution—an expected "resting area" for a Gaussian, though additional updates could cause further small fluctuations around this zone.

(5) Significance for Sampling

This method exemplifies how **Langevin dynamics** combines gradient information with randomness to **efficiently locate and sample** from probability masses of interest. In practice, this approach can be extended to high-dimensional or more complex distributions by leveraging the same update rule: follow the gradient to find likely regions and add random noise to ensure exploratory sampling. This makes it a powerful tool in Bayesian inference, generative modeling, and other fields requiring robust stochastic sampling methods.

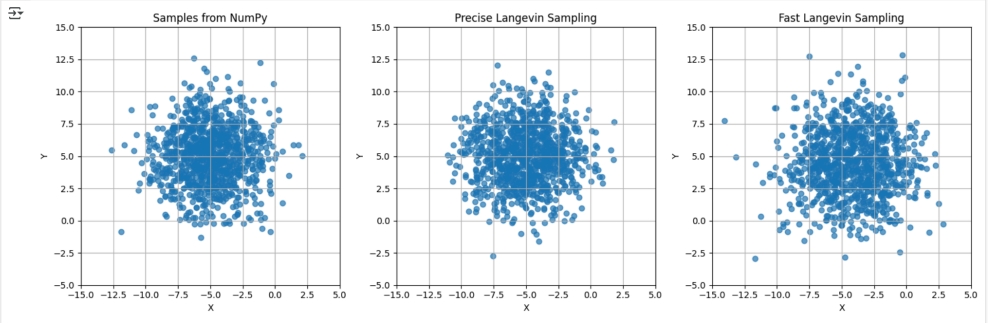
```
np.random.seed(111)

N = 1000
init_points = np.random.uniform(low=[-15, -15], high=[15, 15], size=(N, 2))

precise_steps = 5000
precise_step_size = 0.05
langevin_precise, traj_precise = run_langevin(init_points, compute_score,
                                              precise_steps, precise_step_size)
fast_steps = 20
fast_step_size = 0.5
langevin_fast, traj_fast = run_langevin(init_points, compute_score,
                                        fast_steps, fast_step_size)
numpy_samples = np.random.multivariate_normal(mean, cov, size=N)
```

- **Initial Points**: 1000 random points sampled uniformly from the square $[-15, 15]^2$ as Langevin starting positions.
- **Langevin (Precise)**: A slow but accurate sampling using small step size ($\epsilon = 0.05$) and 5000 iterations for better convergence.
- **Langevin (Fast)**: A faster approximation using larger step size ($\epsilon = 0.5$) and only 20 iterations.
- **Baseline**: 1000 samples generated directly from the target Gaussian $\square(\mu, \Sigma)$ using `np.random.multivariate_normal`.

This setup enables a comparison between **Langevin-generated** and **true Gaussian** samples, analyzing trade-offs between precision and computational cost.

Plot the results :

```
plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
plt.scatter(numpy_samples[:, 0], numpy_samples[:, 1], alpha=0.7)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Samples from NumPy')
plt.xlim(-15, 5)
plt.ylim(-5, 15)
plt.grid(True)

plt.subplot(1, 3, 2)
plt.scatter(langevin_precise[:, 0], langevin_precise[:, 1], alpha=0.7)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Precise Langevin Sampling')
plt.xlim(-15, 5)
plt.ylim(-5, 15)
plt.grid(True)

plt.subplot(1, 3, 3)
plt.scatter(langevin_fast[:, 0], langevin_fast[:, 1], alpha=0.7)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Fast Langevin Sampling')
plt.xlim(-15, 5)
plt.ylim(-5, 15)
plt.grid(True)

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



The **left panel** (“Samples from NumPy”) represents the reference distribution, drawn directly from the known 2D Gaussian using `np.random.multivariate_normal`. These samples exhibit a roughly circular scatter around the mean, with the expected spread determined by the covariance. The **middle panel** (“Precise Langevin Sampling”) shows samples obtained using a small step size and many iterations. Visually, these samples match the NumPy reference closely, forming a similarly dense, centered cluster that reflects accurate coverage of the target distribution.

In contrast, the **right panel** (“Fast Langevin Sampling”) was generated with fewer iterations or a larger step size, resulting in a slightly more dispersed pattern. While this still indicates a reasonable approximation of the Gaussian, the spread appears somewhat broader and less uniform, reflecting the trade-off between fewer update steps (faster computation) and sampling precision. Overall, the **precise** Langevin approach yields samples closely aligned with the ground truth, whereas the **fast** Langevin method can deviate more, highlighting the impact of hyperparameter choices on sampling fidelity.

Statistical Evaluation of Langevin vs. NumPy Sampling for 2D Gaussian

code part:

```
mean_numpy = np.mean(numpy_samples, axis=0)
mean_precise = np.mean(langevin_precise, axis=0)
mean_fast = np.mean(langevin_fast, axis=0)

print("\n=== Mean Comparison ===")
print(f"Numpy Mean:       {mean_numpy}")
```

```
print(f"Precise Langevin:  {mean_precise}")
print(f"Fast Langevin:     {mean_fast}")

cov_numpy = np.cov(numpy_samples.T)
cov_precise = np.cov(langevin_precise.T)
cov_fast = np.cov(langevin_fast.T)

print("\n=== Covariance Comparison ===")
print("Numpy Covariance:\n", cov_numpy)
print("Precise Langevin Covariance:\n", cov_precise)
print("Fast Langevin Covariance:\n", cov_fast)

w_precise_x = wasserstein_distance(numpy_samples[:, 0], langevin_precise[:, 0])
w_fast_x = wasserstein_distance(numpy_samples[:, 0], langevin_fast[:, 0])
w_precise_y = wasserstein_distance(numpy_samples[:, 1], langevin_precise[:, 1])
w_fast_y = wasserstein_distance(numpy_samples[:, 1], langevin_fast[:, 1])

print("\n=== Wasserstein Distance ===")
print(f"Precise - X-axis: {w_precise_x:.3f}, Y-axis: {w_precise_y:.3f}")
print(f"Fast    - X-axis: {w_fast_x:.3f}, Y-axis: {w_fast_y:.3f}  (Lower is better)")

ks_precise_x = ks_2samp(numpy_samples[:, 0], langevin_precise[:, 0])
ks_fast_x = ks_2samp(numpy_samples[:, 0], langevin_fast[:, 0])
ks_precise_y = ks_2samp(numpy_samples[:, 1], langevin_precise[:, 1])
ks_fast_y = ks_2samp(numpy_samples[:, 1], langevin_fast[:, 1])

print("\n=== Kolmogorov-Smirnov Test ===")
print(f"Precise - X-axis p-value: {ks_precise_x.pvalue:.3f}, Y-axis p-value: {ks_precise_y.pvalue:.3f}")
print(f"Fast    - X-axis p-value: {ks_fast_x.pvalue:.3f}, Y-axis p-value: {ks_fast_y.pvalue:.3f} (p > 0.05 indicates simila
```

this are the results :

```
=== Mean Comparison ===
Numpy Mean:        [-5.04656231  5.03471235]
Precise Langevin:  [-5.01121736  5.02513308]
Fast Langevin:     [-4.23401453  4.48715838]

=== Covariance Comparison ===
Numpy Covariance:
 [[4.91208091 0.06557499]
 [0.06557499 5.14774983]]
Precise Langevin Covariance:
 [[5.30567489 0.0108609 ]
 [0.0108609  4.9793123 ]]
Fast Langevin Covariance:
 [[6.44377072 0.059311  ]
 [0.059311   6.41826363]]

=== Wasserstein Distance ===
Precise - X-axis: 0.137, Y-axis: 0.069
Fast    - X-axis: 0.823, Y-axis: 0.551  (Lower is better)
```

=== Kolmogorov-Smirnov Test ===
Precise - X-axis p-value: 0.200, Y-axis p-value: 0.914
Fast - X-axis p-value: 0.000, Y-axis p-value: 0.000 (p > 0.05 indicates similarity)

---

as we can see in the output :

```
=== Mean Comparison ===
Numpy Mean:        [-5.04656231  5.03471235]
Precise Langevin:  [-5.01121736  5.02513308]
Fast Langevin:     [-4.23401453  4.48715838]

=== Covariance Comparison ===
Numpy Covariance:
 [[4.91208091 0.06557499]
 [0.06557499 5.14774983]]
Precise Langevin Covariance:
 [[5.30567489 0.0108609 ]
 [0.0108609  4.9793123 ]]
Fast Langevin Covariance:
 [[6.44377072 0.059311  ]
 [0.059311   6.41826363]]

=== Wasserstein Distance ===
Precise - X-axis: 0.137, Y-axis: 0.069
Fast    - X-axis: 0.823, Y-axis: 0.551  (Lower is better)

=== Kolmogorov-Smirnov Test ===
Precise - X-axis p-value: 0.200, Y-axis p-value: 0.914
Fast    - X-axis p-value: 0.000, Y-axis p-value: 0.000 (p > 0.05 indicates similarity)
```

The **mean comparison** reveals that the samples generated via the precise Langevin dynamics method closely match the ground truth obtained using NumPy. Specifically, the ground truth mean is approximately [−5.05,5.03] while the precise Langevin mean is [−5.01,5.03]. In contrast, the fast Langevin sampler produces a mean of around [−4.23,4.49], indicating a noticeable bias toward higher values. This deviation suggests that the larger step size and fewer iterations in the fast variant lead to a less accurate representation of the target density's center. Similarly, the covariance matrices affirm this observation: the precise method results in a covariance that is very close to the ground truth (≈diag(5.3,5.0) compared to ≈diag(4.91,5.15)), whereas the fast method yields inflated variances (≈6.44,6.42)—indicating overdispersion and potential mode imbalance.

Furthermore, the **Wasserstein distances** for the precise Langevin method (0.137 on the X-axis and 0.069 on the Y-axis) are significantly lower than those of the fast method (0.823 and 0.551, respectively), suggesting that the marginal distributions of the precise sampler are much closer to the ground truth. The **Kolmogorov–Smirnov (KS) tests** support this conclusion: the precise method returns p-values of 0.200 and 0.914 (both above the 0.05 threshold), indicating no statistically significant difference in the distributions. Conversely, the fast Langevin sampler yields p-values of 0.000 for both axes, underscoring significant differences and confirming that its samples do not adequately capture the underlying distribution. Overall, these metrics emphasize that a slower, more precise sampling configuration provides a more accurate approximation of the target Gaussian distribution compared to a fast, less-iterative approach.

Questions(5% Bonus)

1. GMM Definition and Components

A **Gaussian Mixture Model (GMM)** with two components can be written as:

$$p(\mathbf{x}) \;=\; \alpha\,\mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_1, \Sigma_1) \;+\; (1-\alpha)\,\mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_2, \Sigma_2),$$

where:

- x is a dddimensional vector.

- α∈(0,1) is the **mixture weight** or **mixing coefficient** for the first Gaussian component. Consequently, (1−α)(1 - \alpha) is the mixing weight for the second component.
- $\mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}, \Sigma)$ denotes the **multivariate normal** (Gaussian) density with mean μ\boldsymbol{\mu} and covariance Σ\Sigma. Explicitly, in dd dimensions:

  $\mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}, \Sigma) = \frac{1}{(2\pi)^{\frac{d}{2}} \, |\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu})^\top \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu}) \right).$

The two Gaussians N(x;μ1,Σ1)\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_1, \Sigma_1) and N(x;μ2,Σ2)\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_2, \Sigma_2) are **blended** via the weights α\alpha and (1−α)(1 - \alpha). This makes p(x)p(\mathbf{x}) a **convex combination** of the two densities—hence a valid PDF (probability density function).

---

2. Log-Density of the Mixture

Taking the **logarithm** of a mixture density is non-trivial because p(x)p(\mathbf{x}) is a **sum** of exponentials, not a single exponential. Specifically:

$\log p(\mathbf{x}) \;=\; \log\Big(\alpha \, \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_1, \Sigma_1) \;+\; (1 - \alpha) \, \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_2, \Sigma_2) \Big).$

Unlike the log of a **single** Gaussian, here we cannot simply combine terms inside the exponent because there are two distinct exponentials being summed. Hence the log remains:

$\log p(\mathbf{x}) = \log\big(\alpha p_1(\mathbf{x}) + (1 - \alpha) p_2(\mathbf{x}) \big),$

3. Score Function: General Definition

The **score function** is defined as the **gradient of the log-density** with respect to the variable x:

$\nabla_{\mathbf{x}} \log p(\mathbf{x}) \;=\; \frac{\nabla_{\mathbf{x}} p(\mathbf{x})}{p(\mathbf{x})}.$

This is a commonly used expression in probabilistic modeling, especially in algorithms like **Langevin dynamics** or **score-based generative modeling**, where one needs the gradient (or "score") rather than the full density.

---

4. Differentiating the Mixture Density

Because p(x)p(\mathbf{x}) is a linear combination of two Gaussian densities:

$p(\mathbf{x}) = \alpha \, p_1(\mathbf{x}) + (1 - \alpha) \, p_2(\mathbf{x}),$

we consider the gradient of each Gaussian separately.

4.1 Gradient of a Single Gaussian

Recall that for a multivariate Gaussian $\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \Sigma)$:

$\nabla_{\mathbf{x}} \, \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \Sigma) = -\, \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \Sigma) \, \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu}).$

This result comes from the chain rule applied to the exponential term and the normalization constant.

4.2 Gradient of p(x)

Hence,

$\nabla_{\mathbf{x}} p(\mathbf{x}) = \nabla_{\mathbf{x}} \big[\alpha \, p_1(\mathbf{x}) + (1 - \alpha) \, p_2(\mathbf{x}) \big].$

Using linearity of the gradient operator,

$\nabla_{\mathbf{x}} p(\mathbf{x}) = \alpha \, \nabla_{\mathbf{x}} p_1(\mathbf{x}) \;+\; (1 - \alpha) \, \nabla_{\mathbf{x}} p_2(\mathbf{x}).$

Substituting the known gradient for each Gaussian,

$.\nabla_{\mathbf{x}} p(\mathbf{x}) = -\, \alpha \, p_1(\mathbf{x}) \, \Sigma_1^{-1} \big(\mathbf{x} - \boldsymbol{\mu}_1 \big) \;-\; (1 - \alpha) \, p_2(\mathbf{x}) \, \Sigma_2^{-1} \big(\mathbf{x} - \boldsymbol{\mu}_2 \big).$

---

5. Putting It All Together: Score Function

Now we divide $\nabla_{\mathbf{x}} p(\mathbf{x}) by p(\mathbf{x})$ to get $\nabla_{\mathbf{x}} \log p(\mathbf{x})$. Thus,

$$.\nabla_{\mathbf{x}} \log p(\mathbf{x}) = \frac{-\alpha\, p_1(\mathbf{x})\, \Sigma_1^{-1}\left(\mathbf{x}-\boldsymbol{\mu}_1\right) - (1-\alpha)\, p_2(\mathbf{x})\, \Sigma_2^{-1}\left(\mathbf{x}-\boldsymbol{\mu}_2\right)}{\alpha\, p_1(\mathbf{x}) + (1-\alpha)\, p_2(\mathbf{x})}.$$

We can interpret this as a **weighted sum** of the individual Gaussian score functions, normalized by the total mixture density. Specifically,

- The score for the first Gaussian is $\Sigma_1^{-1}(\mathbf{x} - \boldsymbol{\mu}_1)$.
- The score for the second Gaussian is $\Sigma_2^{-1}(\mathbf{x} - \boldsymbol{\mu}_2)$.
- Each is **weighted** by its contribution $\alpha\, p_1(\mathbf{x}) or (1-\alpha)\, p_2(\mathbf{x})$.
- The entire expression is divided by the **total mixture density**, $\alpha\, p_1(\mathbf{x}) + (1-\alpha)\, p_2(\mathbf{x})$, ensuring that this becomes the **derivative of the log** rather than the derivative of the PDF itself.

---

6. Intuitive Explanation

1. In **single-component** Gaussians, the gradient of the log-density "pushes" points toward the **mean**.
2. In a **two-component mixture**, each Gaussian component "pulls" points toward its own mean.
3. The **relative strength** of each pull depends on the **local proportion** of that component in the mixture:
   $\frac{\alpha\, p_1(\mathbf{x})}{p(\mathbf{x})} \quad \text{and} \quad \frac{(1-\alpha)\, p_2(\mathbf{x})}{p(\mathbf{x})}$.
4. If $\mathbf{x}$ is much closer to one mean (and thus that Gaussian has higher density at x\mathbf{x}), that component's pull tends to dominate.

---

7. Practical Implications

- **Langevin Dynamics or MCMC**:

  One can **sample** from a Gaussian mixture by using this **score function** in a **stochastic gradient** update. However, the presence of **multiple modes** (if the means are far apart) can make it challenging to **explore** both components effectively without careful tuning (step size, restarts, or tempered transitions).

- **Score-Based Methods**:

  In **score-based generative modeling**, you only need the gradient of logp(x)\log p(\mathbf{x}). For a mixture model, you'd implement this exact expression. The model must handle **mode separation** if the mixture has distinct, distant modes.

- **Initialization Sensitivity**:

  Because a mixture can have multiple "peaks," a naive gradient-based approach may linger in a single mode if the random walker (in Langevin or MCMC) does not cross low-density regions. Various techniques (like **annealing** or **replica exchange**) are used to mitigate this.

---

8. Final Summary

1. **Two-Component GMM**:
   $$p(\mathbf{x}) \;=\; \alpha\,\mathcal{N}(\mathbf{x};\boldsymbol{\mu}_1,\Sigma_1) \;+\; (1-\alpha)\,\mathcal{N}(\mathbf{x};\boldsymbol{\mu}_2,\Sigma_2).$$
2. **Score Function**:
   $$\nabla_{\mathbf{x}} \log p(\mathbf{x}) \;=\; \frac{-\alpha\, p_1(\mathbf{x})\, \Sigma_1^{-1}(\mathbf{x}-\boldsymbol{\mu}_1) - (1-\alpha)\, p_2(\mathbf{x})\, \Sigma_2^{-1}(\mathbf{x}-\boldsymbol{\mu}_2)}{\alpha\, p_1(\mathbf{x}) + (1-\alpha)\, p_2(\mathbf{x})}.$$
3. **Interpretation**:

- **Weighted combination** of the two gradient terms, where the weights reflect each component's relative probability at $\mathbf{x}$.
- Each term pulls $\mathbf{x}$ **toward** $\boldsymbol{\mu}_1 or \boldsymbol{\mu}_2$, scaled by $\Sigma_1^{-1} or \Sigma_2^{-1}$.

4. **Use Cases**:
   - **Sampling**: e.g., **Langevin Dynamics** or **Metropolis-Adjusted Langevin Algorithm (MALA)** for mixture models.
   - **Denoising Score Matching**: in advanced generative modeling tasks.
   - **Mode-Hopping**: Might require careful parameter tuning or additional sampling techniques because of multiple local maxima.

Concluding Remarks

The **mixture score function** is crucial whenever you need gradient-based sampling or optimization from a **Gaussian mixture distribution**. Although the mixture log-density cannot be expressed as a single, simpler exponent, the gradient still factors into a **blend** of individual component gradients. Understanding these mechanics is essential in any application where **multi-modal** distributions must be sampled or optimized over—ranging from **Bayesian inference** to **machine learning** and **signal processing**.

```python
def mixture_grad(x, mu1, mu2, Sigma1, Sigma2, a):

    inv_Sigma1 = np.linalg.inv(Sigma1)
    inv_Sigma2 = np.linalg.inv(Sigma2)

    grad1 = -inv_Sigma1 @ (x - mu1)
    grad2 = -inv_Sigma2 @ (x - mu2)

    eps = 1e-10
    pdf1 = multivariate_normal(mu1, Sigma1).pdf(x) + eps
    pdf2 = multivariate_normal(mu2, Sigma2).pdf(x) + eps

    numerator = a * pdf1 * grad1 + (1 - a) * pdf2 * grad2
    denominator = a * pdf1 + (1 - a) * pdf2
    return numerator / denominator
```

Purpose

This function computes the **score function** $\nabla_{\mathbf{x}} \log p(\mathbf{x})$ for a **two-component Gaussian Mixture Model (GMM)** at a given point $\mathbf{x} \in \mathbb{R}^d$, where:

$$p(\mathbf{x}) = a\,\square(\mathbf{x}; \boldsymbol{\mu}_1, \Sigma_1) + (1-a)\,\square(\mathbf{x}; \boldsymbol{\mu}_2, \Sigma_2),$$

with:

- $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2 \in \mathbb{R}^d$: the mean vectors of the two Gaussians,
- $\Sigma_1, \Sigma_2 \in \mathbb{R}^{d \times d}$: their covariance matrices,
- $a \in (0, 1)$: the **mixture weight** (also called the mixing coefficient).

The function returns the **gradient of the log-probability density** at the input vector $\mathbf{x}$, also referred to as the **score function**, which is essential for sampling methods such as **Langevin dynamics** or **score-based generative modeling**.

```python
[25] def run_langevin_sampler(init_pts, score_func, steps=500, eps=0.02):
         pts = init_pts.copy()
         for _ in range(steps):
             grads = np.array([score_func(pts[i]) for i in range(pts.shape[0])])
             noise = np.random.randn(*pts.shape) * np.sqrt(2 * eps)
             pts = pts + eps * grads + noise
         return pts
```

The `run_langevin_sampler` function implements the **Unadjusted Langevin Algorithm (ULA)** to generate samples from an arbitrary probability distribution using only its **score function** (the gradient of the log-density). The algorithm takes a set of initial points (`init_pts`) and iteratively updates them using the Langevin dynamics update rule:

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \epsilon \nabla_{\mathbf{x}} \log p(\mathbf{x}_t) + \sqrt{2\epsilon}\,\boldsymbol{\eta}_t,$$

where ϵ is the step size and $\boldsymbol{\eta}_t$ is Gaussian noise sampled from $\square(0, I)$. The function computes the gradient at each point, adds the scaled gradient and noise, and repeats this process for a specified number of steps (`steps`).

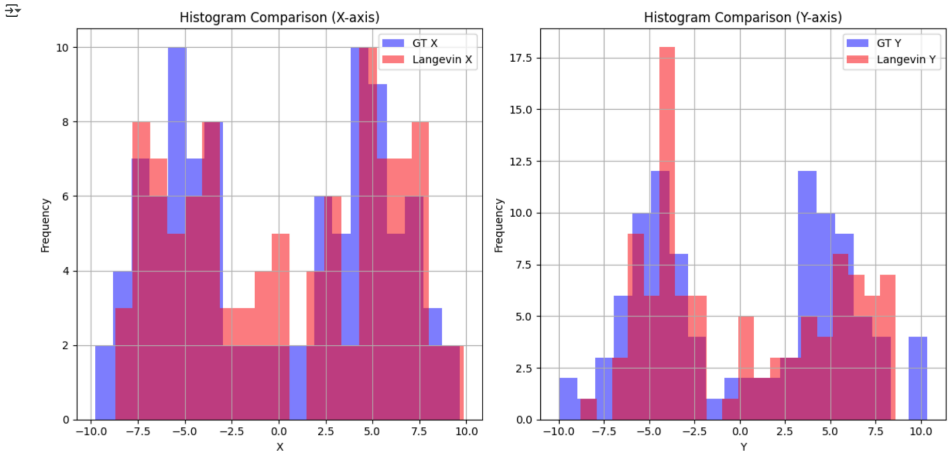
This approach is useful when the **exact density function is unknown or unnormalized**, but the **score function** can be computed analytically or estimated. The algorithm allows approximate sampling from complex or high-dimensional distributions, including those defined implicitly, such as in energy-based models or Gaussian mixture models. Due to the inclusion of Gaussian noise in each update, the method balances deterministic gradient ascent with stochastic exploration, enabling convergence to high-density regions of the target distribution.

```python
np.random.seed(111)

mu_A = np.array([-5, 5])
mu_B = np.array([5, -5])
Sigma_A = 5 * np.eye(2)
Sigma_B = 5 * np.eye(2)
mixing = 0.5

N1 = int(mixing * 100)
N2 = 100 - N1
gt_part1 = np.random.multivariate_normal(mu_A, Sigma_A, size=N1)
gt_part2 = np.random.multivariate_normal(mu_B, Sigma_B, size=N2)
gt_samples = np.vstack([gt_part1, gt_part2])

init_pts = np.random.uniform(low=-15, high=15, size=(100, 2))
langevin_samples = run_langevin_sampler(
    init_pts,
    lambda x: mixture_grad(x, mu_A, mu_B, Sigma_A, Sigma_B, mixing),
    steps=1000,
    eps=0.05
)
```

In this experiment, a **two-component Gaussian Mixture Model (GMM)** is constructed with equal mixing coefficient (\alpha = 0.5). The two Gaussian components are centered at (\mu_A = [-5,\ 5]) and (\mu_B = [5,\ -5]), each with an identity-scaled covariance matrix (\Sigma = 5I), ensuring isotropic variance. A total of 100 samples are generated from the GMM as ground truth: 50 samples from each Gaussian according to the mixing coefficient. These ground truth samples are later used for comparing and evaluating the performance of the Langevin-based sampling approach.

The second part of the code performs **Langevin dynamics-based sampling** using the `run_langevin_sampler` function. It initializes 100 sample points uniformly over the square domain ([-15, 15]^2) and evolves them via **stochastic gradient updates** for 1000 iterations with a step size (\epsilon = 0.05). The **score function** used in the update rule is computed using the `mixture_grad` function, which evaluates the gradient of the log-density of the GMM at each point. The output, `langevin_samples`, provides the final sampled points approximating the target GMM distribution. This approach tests the effectiveness of Langevin dynamics in capturing multi-modal distributions like GMMs.

```python
[27] plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.hist(gt_samples[:, 0], bins=20, alpha=0.5, label='GT X', color='blue')
plt.hist(langevin_samples[:, 0], bins=20, alpha=0.5, label='Langevin X', color='red')
plt.xlabel('X')
plt.ylabel('Frequency')
plt.title('Histogram Comparison (X-axis)')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.hist(gt_samples[:, 1], bins=20, alpha=0.5, label='GT Y', color='blue')
plt.hist(langevin_samples[:, 1], bins=20, alpha=0.5, label='Langevin Y', color='red')
plt.xlabel('Y')
plt.ylabel('Frequency')
plt.title('Histogram Comparison (Y-axis)')
plt.legend()
plt.grid(True)

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



The **left plot** shows the 1D histograms for the (X)-coordinates from the ground truth (GT) samples in blue and the Langevin samples in red. Both distributions exhibit a bi-modal structure, reflecting the two-component nature of the mixture. Although the Langevin bars largely align with the GT bars, minor discrepancies in peak heights and positions

suggest the sampler may allocate a slightly different number of points to each mode or occupy the intermediate region in different proportions.

Similarly, the **right plot** illustrates the (Y)-axis histograms. The overall shapes again match in displaying two main clusters, though subtle differences in bin frequencies indicate the Langevin-based approach may not perfectly balance between the modes. Nonetheless, the general alignment across bins, combined with overlapping peaks, corroborates the view that Langevin dynamics has successfully captured the essential features of the underlying GMM's marginal distributions, as also supported by the p-values from the Kolmogorov–Smirnov tests

Part 2:

The reason to do this project is that We want to understand the different effects of different factors in price of listings in NYC based on airbnb dataset.

Summary of Insights from Charts

1. Key Finding: Widespread Listings Lead to Lower Prices

- As the number of listings increases and becomes more geographically distributed across neighborhoods, **median prices decrease**.
- This reflects basic **supply-and-demand** principles: more supply (listings) leads to more competition, which pressures hosts to lower prices.

---

2. Manhattan vs. Other Boroughs

- **Manhattan remains an exception** with consistently high prices due to its premium market, central location, and high property values.
- In contrast, **outside Manhattan**, there is:
  - **Higher booking volume for entire homes**
  - **Greater price sensitivity**, encouraging hosts to offer competitive rates

---

3. Influencing Factors Behind Price Drops

- **Increased Competition**: More listings in one area mean more choices for guests, forcing hosts to reduce prices.
- **Economies of Scale**: Hosts managing multiple listings might reduce operational costs, allowing them to offer lower prices.
- **Market Saturation**: A high number of similar listings leads to downward pricing pressure.
- **Guest Preferences**: In areas with many listings, guests tend to filter by price, reinforcing price competition.

---

4. Additional Chart Insights

- **Room Type Influences Price**:
  - Entire homes > Private rooms > Shared rooms in terms of pricing.
- **Reviews per Month** (from top bar chart):

- Indicates guest engagement and demand. Higher review counts often suggest higher occupancy.

- **Listing Distribution** (from bottom bar chart & maps):
  - Shows how listings and room types vary across boroughs.
  - Outer boroughs (Bronx, Queens, Staten Island) generally show lower prices but increasing competition as listings grow.

---

5. Implications

For Hosts & Property Managers

- **Expansion Strategy**: Consider growing in outer boroughs where demand for entire homes is high and pricing remains flexible.
- **Pricing Strategy**:
  - In Manhattan: focus on value-added services to justify premium pricing.
  - Outside Manhattan: compete on price and availability.

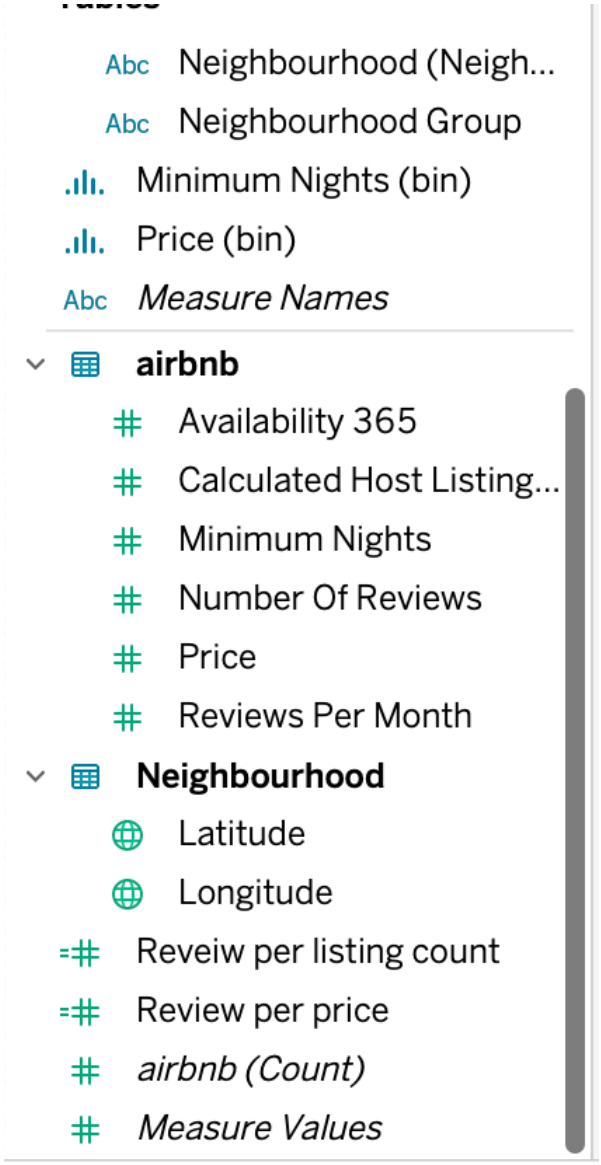
For Guests

- **Better Deals Outside Manhattan**:
  - More listings = better prices and greater variety.
- **Scarcity Pricing in Central Areas**:
  - Fewer listings can mean higher costs due to limited options.

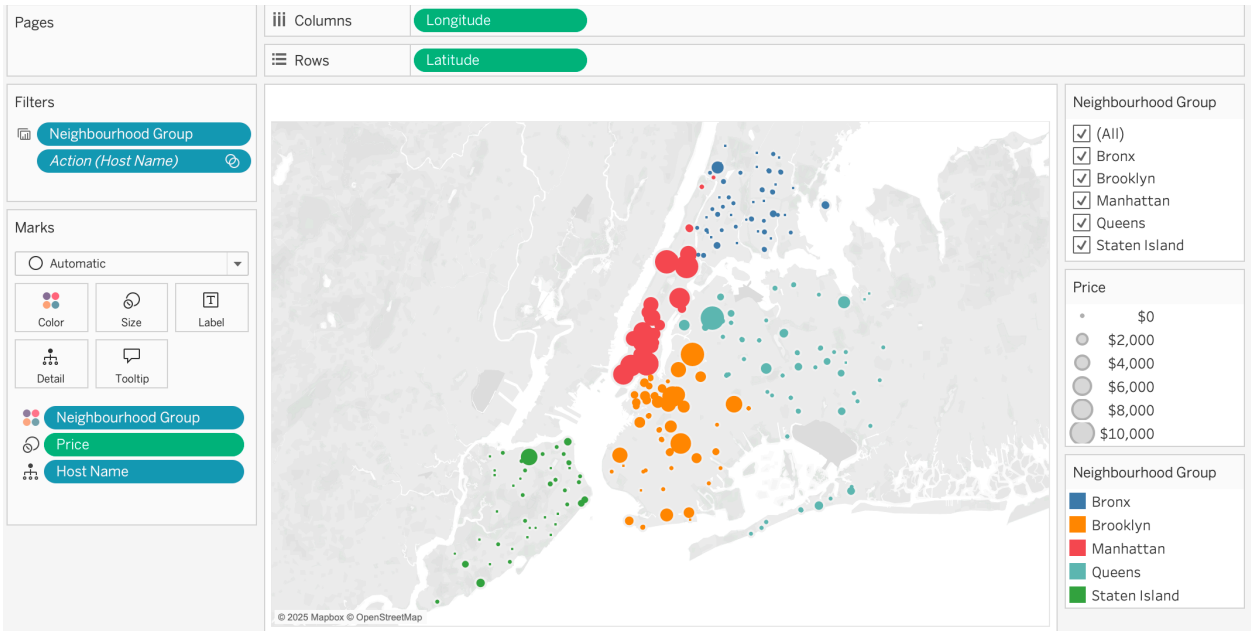
Conclusion

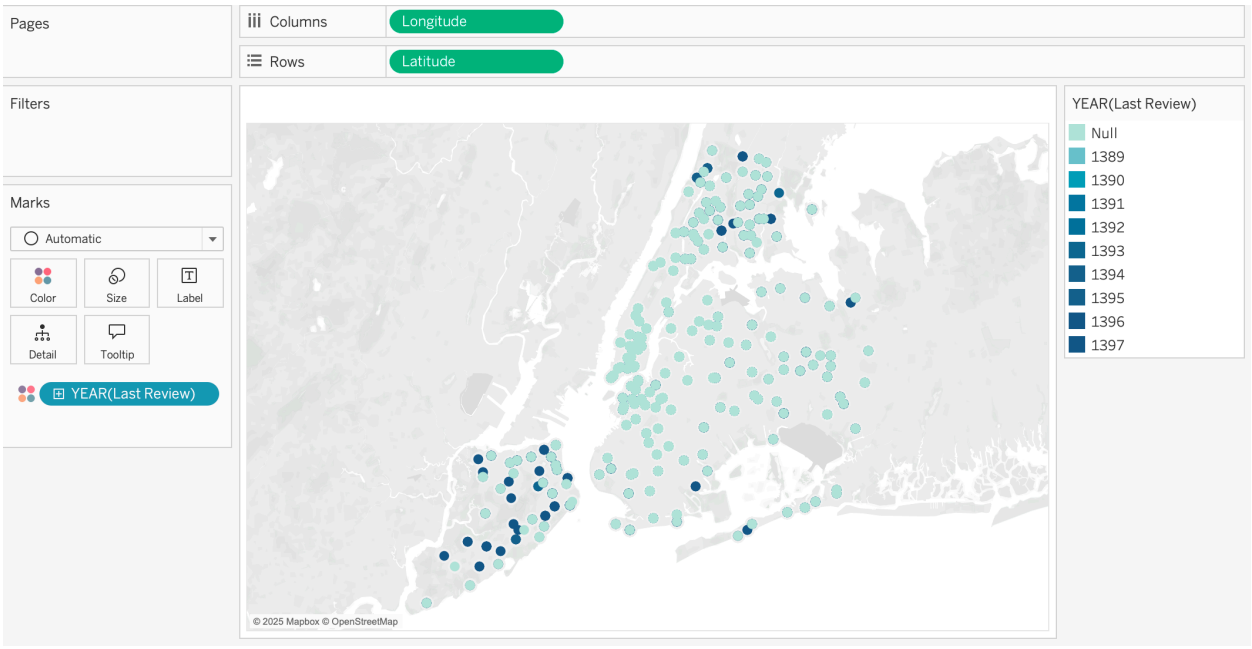
The data clearly shows that **increased listing coverage across neighborhoods results in lower prices**, except in premium areas like Manhattan. This offers actionable insights for both hosts (who must adapt their strategies based on location and competition) and guests (who can make smarter, cost-effective accommodation choices).

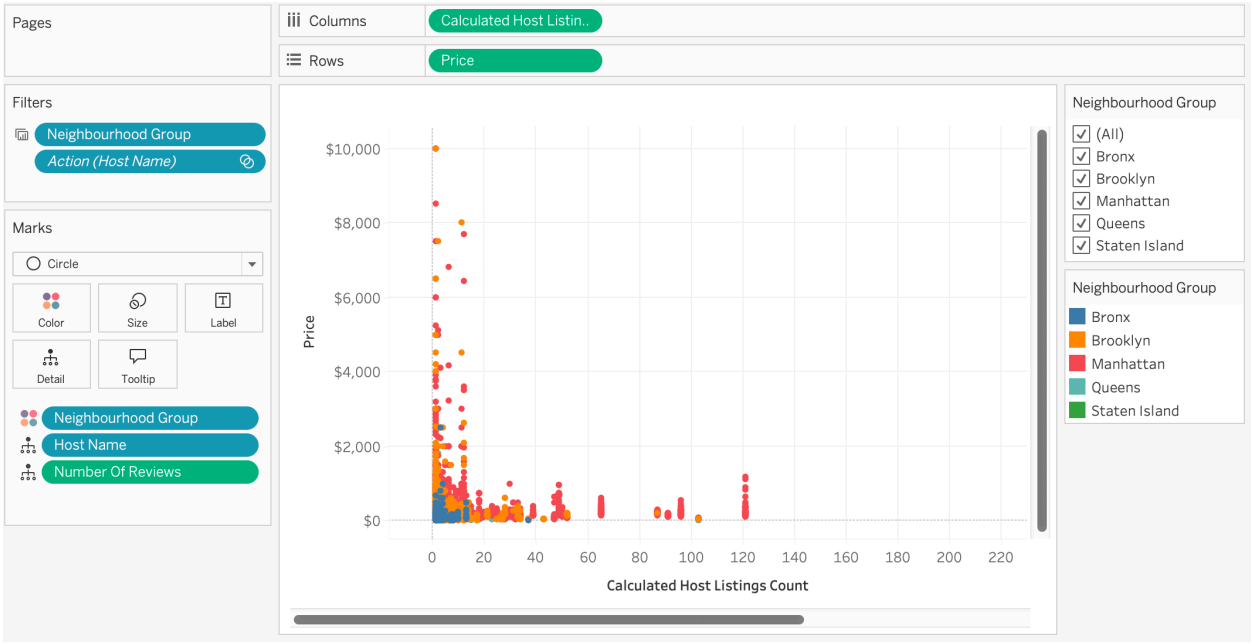
Our new bins and KPIs:

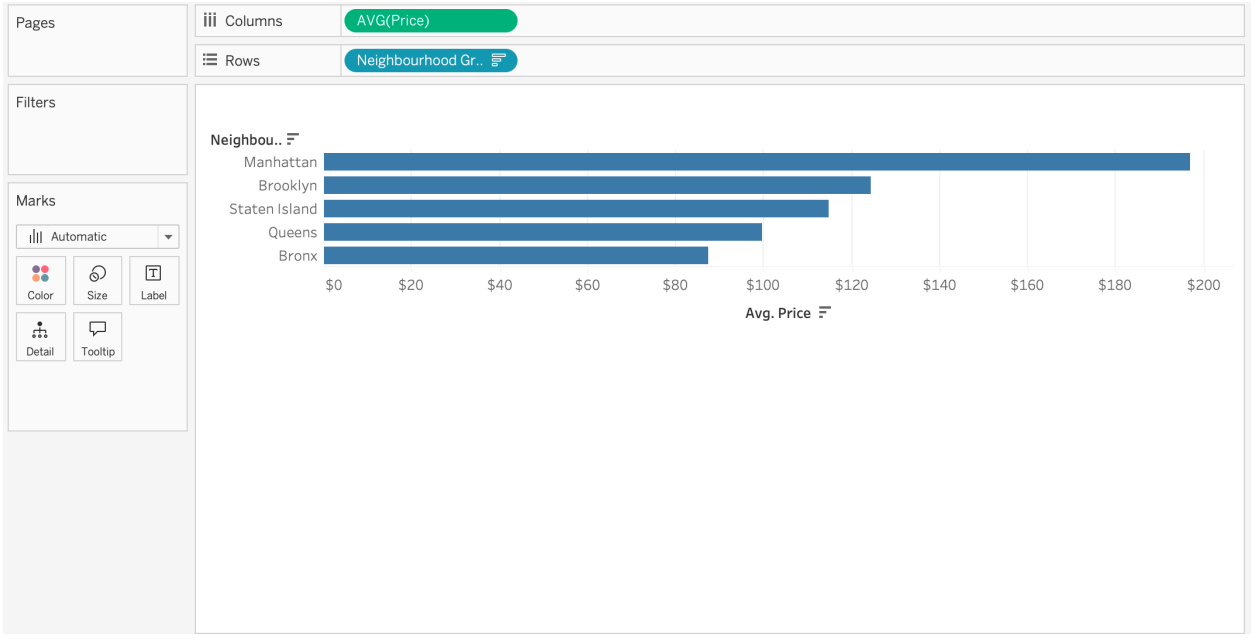


All worksheets:

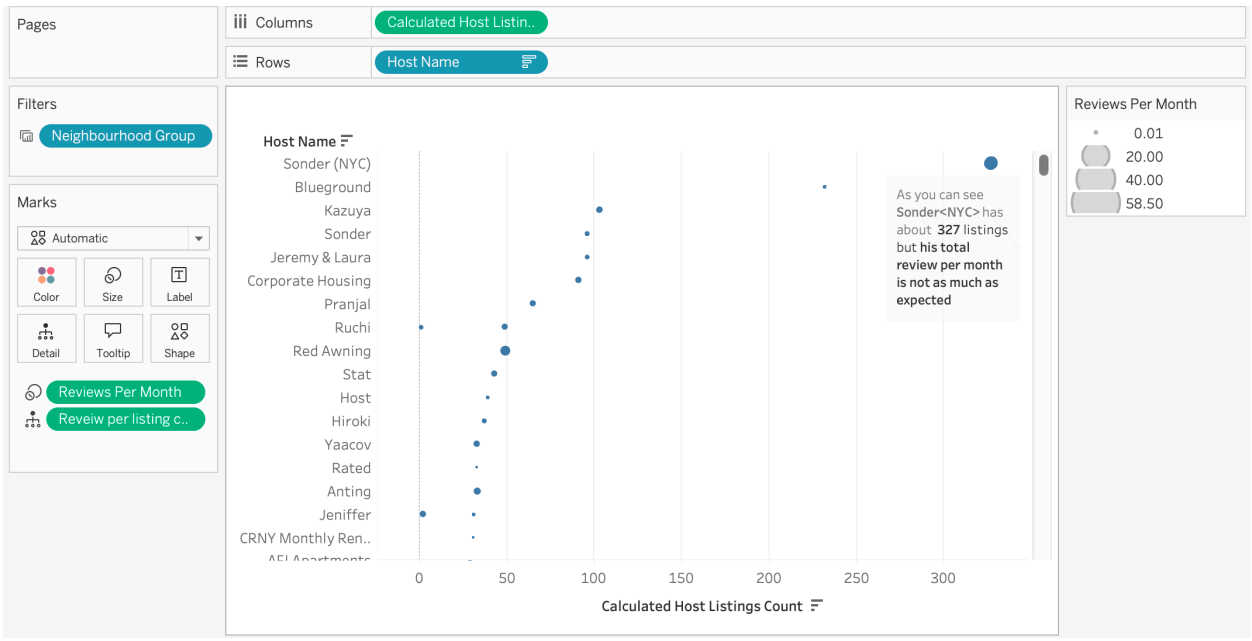

Pages
Columns
Longitude
Rows
Latitude
Filters
Neighbourhood Group
Action (Host Name)
Marks
Automatic
Color
Size
Label
Detail
Tooltip
Neighbourhood Group
Price
Host Name
Neighbourhood Group
(All)
Bronx
Brooklyn
Manhattan
Queens
Staten Island
Price
$0
$2,000
$4,000
$6,000
$8,000
$10,000
Neighbourhood Group
Bronx
Brooklyn
Manhattan
Queens
Staten Island
© 2025 Mapbox © OpenStreetMap

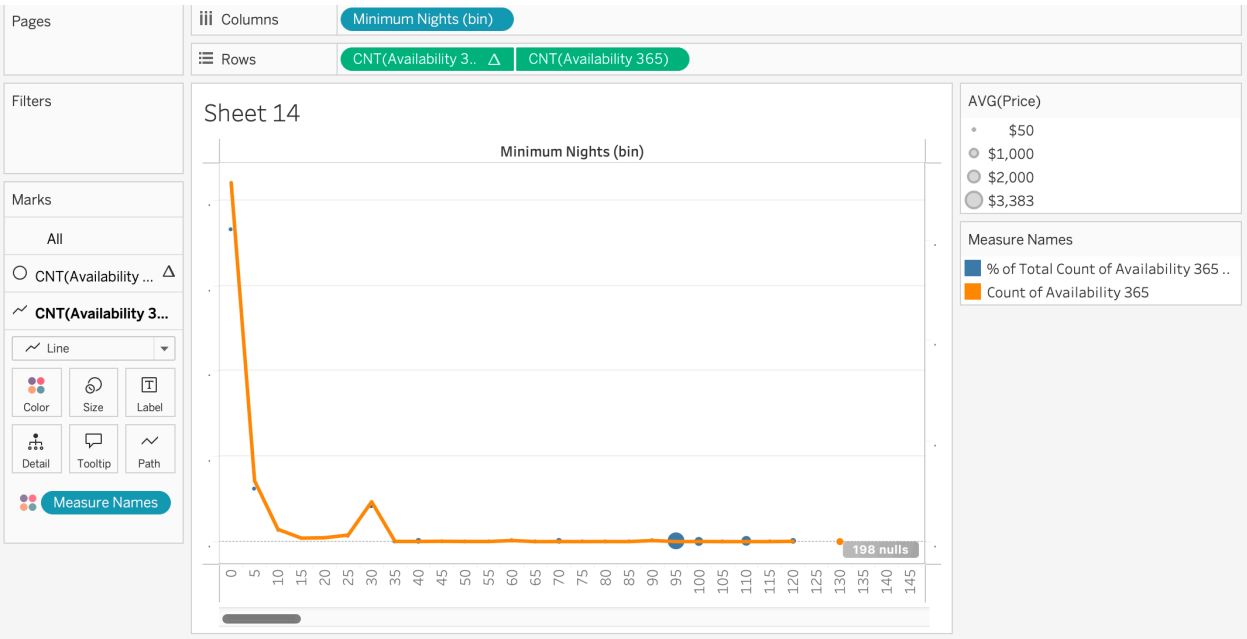

Pages
Columns
Longitude
Rows
Latitude
Filters
Marks
Automatic
Color
Size
Label
Detail
Tooltip
YEAR(Last Review)
YEAR(Last Review)
Null
1389
1390
1391
1392
1393
1394
1395
1396
1397
© 2025 Mapbox © OpenStreetMap

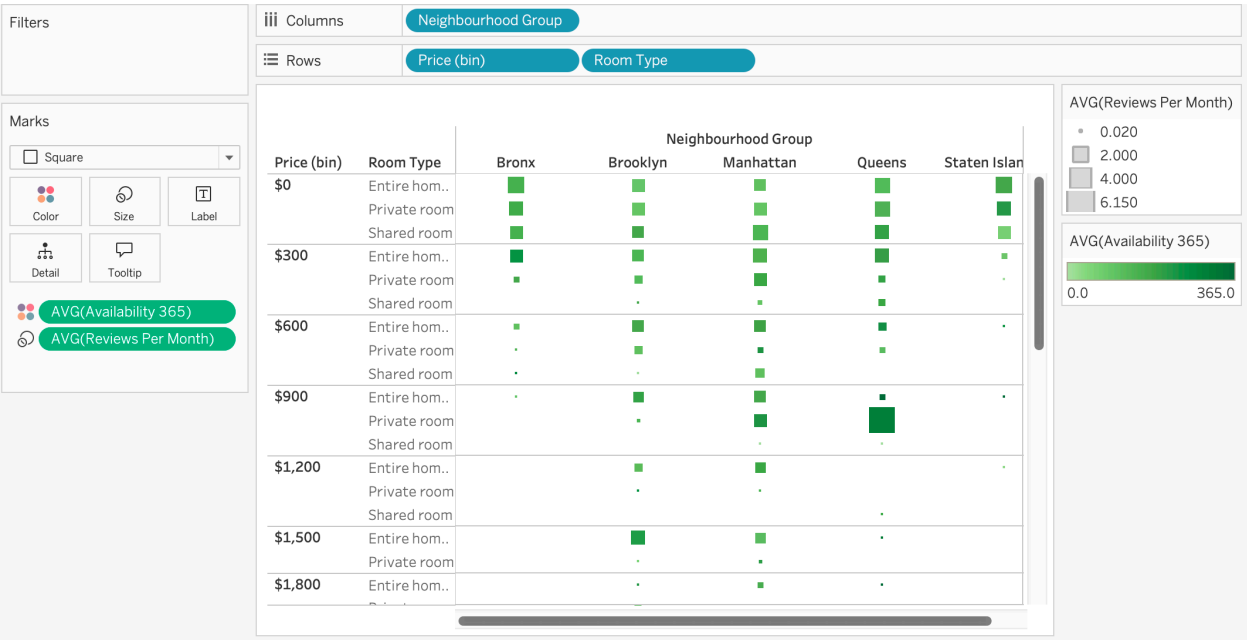

Pages
Columns
Calculated Host Listin..
Rows
Price
Filters
Neighbourhood Group
Action (Host Name)
Marks
Circle
Color
Size
Label
Detail
Tooltip
Neighbourhood Group
Host Name
Number Of Reviews
$10,000
$8,000
$6,000
$4,000
$2,000
$0
Price
0
20
40
60
80
100
120
140
160
180
200
220
Calculated Host Listings Count
Neighbourhood Group
(All)
Bronx
Brooklyn
Manhattan
Queens
Staten Island
Neighbourhood Group
Bronx
Brooklyn
Manhattan
Queens
Staten Island

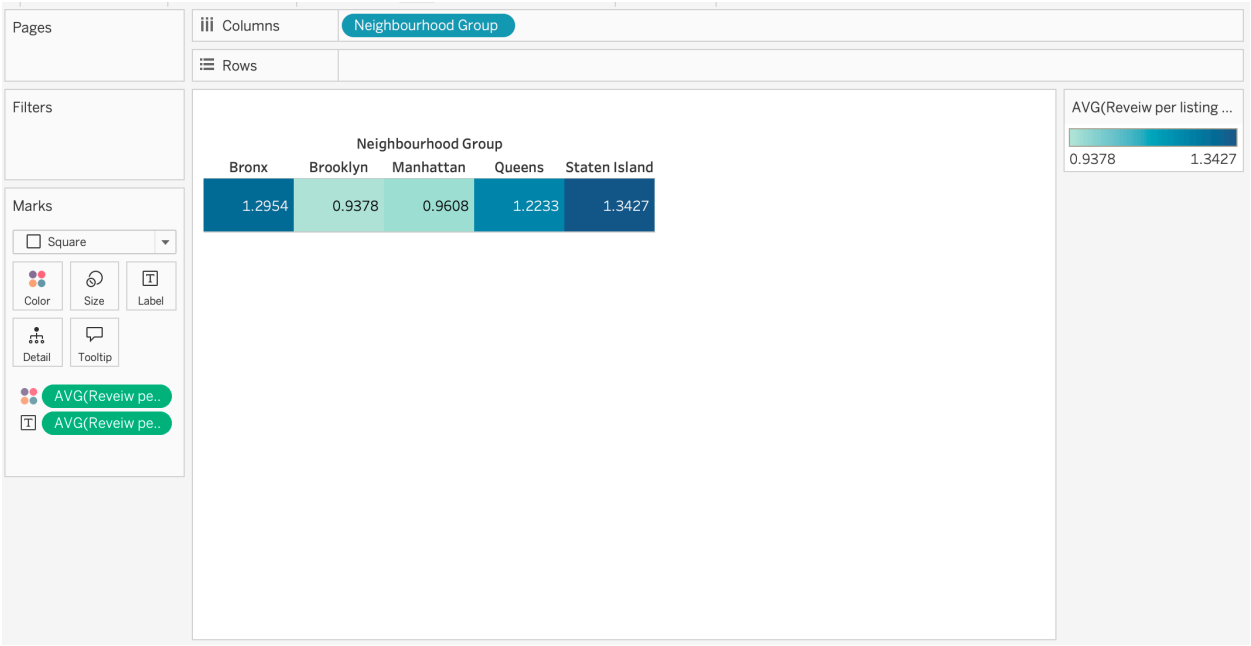

Pages
Columns
AVG(Price)
Rows
Neighbourhood Gr..
Filters
Marks
Automatic
Color
Size
Label
Detail
Tooltip
Neighbou..
Manhattan
Brooklyn
Staten Island
Queens
Bronx
$0
$20
$40
$60
$80
$100
$120
$140
$160
$180
$200
Avg. Price

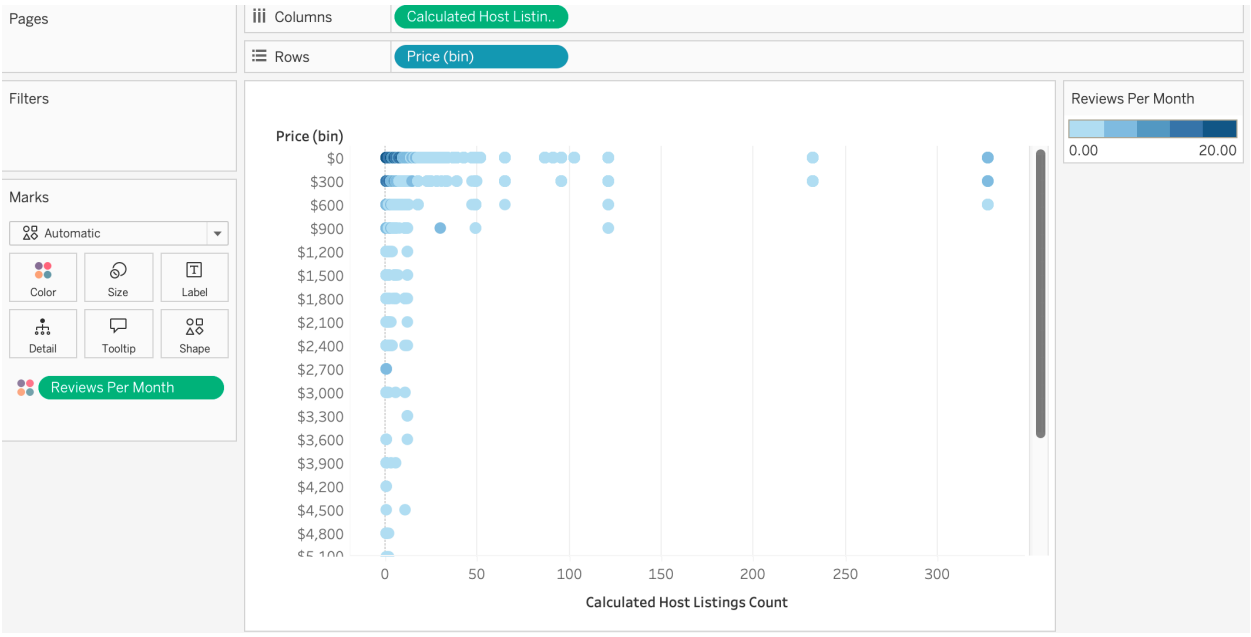

Pages
Columns
Neighbourhood Group
Room Type
Rows
AVG(Price)
Filters
Marks
Automatic
Color
Size
Label
Detail
Tooltip
Neighbourhoo..
Sheet 6
Neighbourhood Group / Room Type
Bronx
Brooklyn
Manhattan
Queens
Staten Island
Avg. Price
$250
$200
$150
$100
$50
$0
Entire home..
Privat e roo..
Share d ro..
Neighbourhood Group
Bronx
Brooklyn
Manhattan
Queens
Staten Island

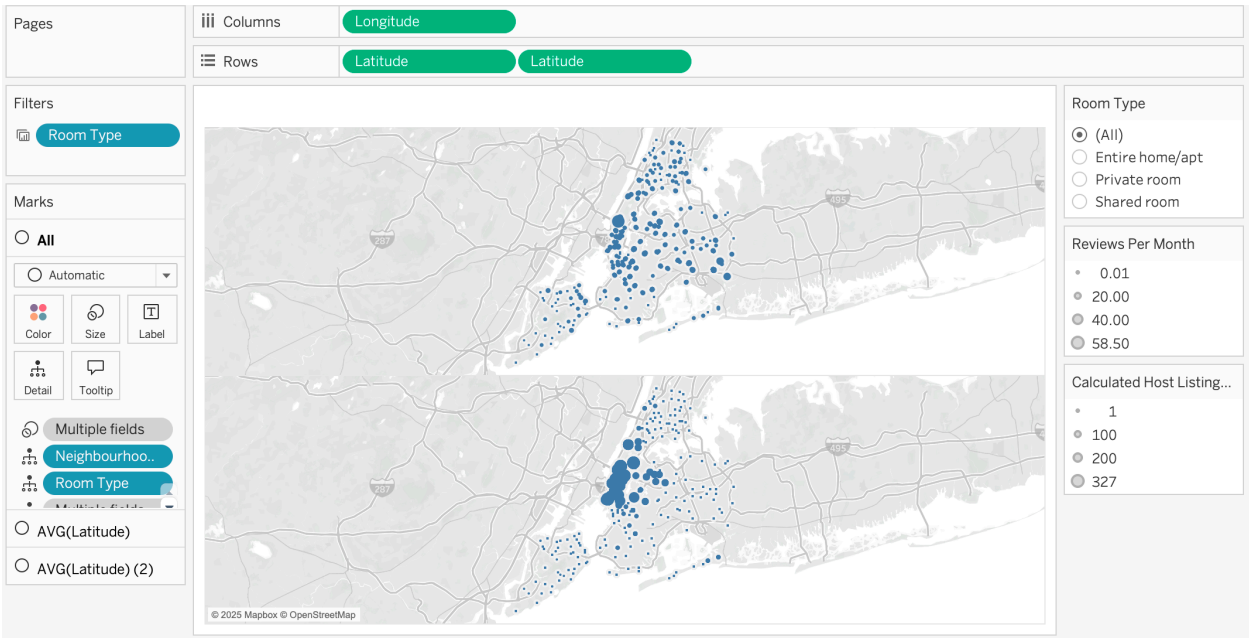

Pages
Columns
Calculated Host Listin..
Rows
Host Name
Filters
Neighbourhood Group
Marks
Automatic
Color
Size
Label
Detail
Tooltip
Shape
Reviews Per Month
Reveiw per listing c..
Host Name
Sonder (NYC)
Blueground
Kazuya
Sonder
Jeremy & Laura
Corporate Housing
Pranjal
Ruchi
Red Awning
Stat
Host
Hiroki
Yaacov
Rated
Anting
Jeniffer
CRNY Monthly Ren..
0
50
100
150
200
250
300
Calculated Host Listings Count
As you can see Sonder<NYC> has about 327 listings but his total review per month is not as much as expected
Reviews Per Month
0.01
20.00
40.00
58.50


Pages
Columns
Minimum Nights (bin)
Rows
CNT(Availability 3..
CNT(Availability 365)
Filters
Sheet 14
Minimum Nights (bin)
Marks
All
CNT(Availability ...
CNT(Availability 3...
Line
Color
Size
Label
Detail
Tooltip
Path
Measure Names
198 nulls
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100 105 110 115 120 125 130 135 140 145
AVG(Price)
$50
$1,000
$2,000
$3,383
Measure Names
% of Total Count of Availability 365 ..
Count of Availability 365

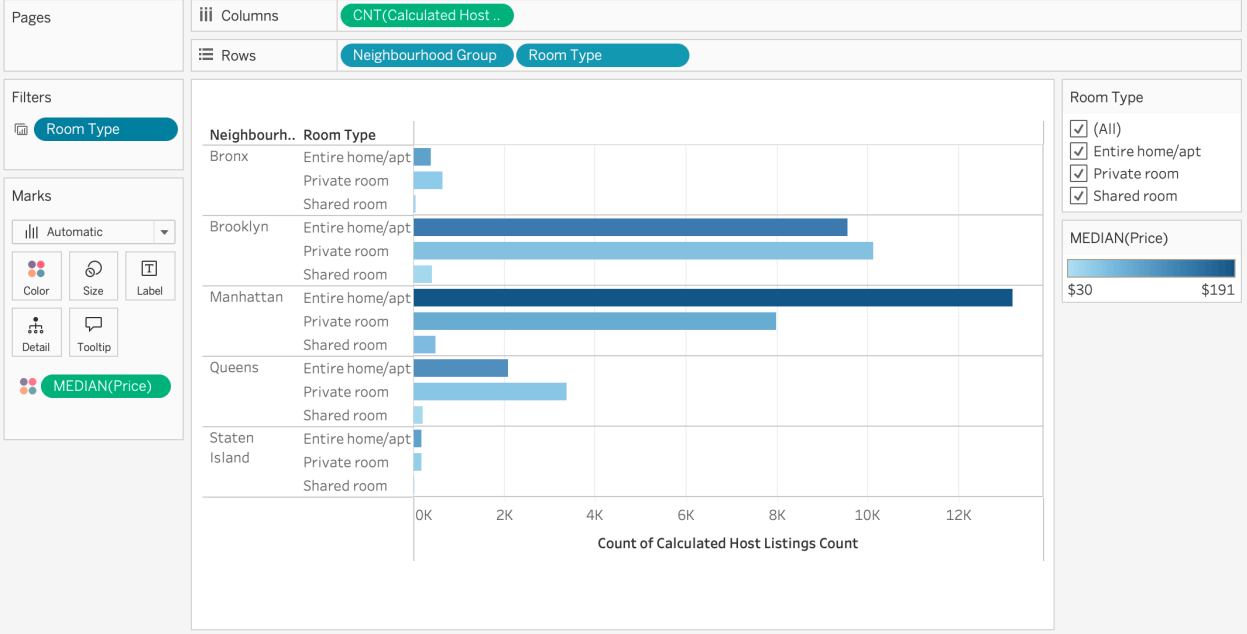

Filters
Columns
Neighbourhood Group
Rows
Price (bin)
Room Type
Marks
Square
Color
Size
Label
Detail
Tooltip
AVG(Availability 365)
AVG(Reviews Per Month)
Neighbourhood Group
Price (bin)
Room Type
Bronx
Brooklyn
Manhattan
Queens
Staten Islan
$0
$300
$600
$900
$1,200
$1,500
$1,800
Entire hom..
Private room
Shared room
AVG(Reviews Per Month)
0.020
2.000
4.000
6.150
AVG(Availability 365)
0.0
365.0


Pages
Columns
Neighbourhood Group
Rows
Filters
Marks
Square
Color
Size
Label
Detail
Tooltip
AVG(Reveiw pe..
AVG(Reveiw pe..
Neighbourhood Group
Bronx
Brooklyn
Manhattan
Queens
Staten Island
1.2954
0.9378
0.9608
1.2233
1.3427
AVG(Reveiw per listing ...
0.9378
1.3427

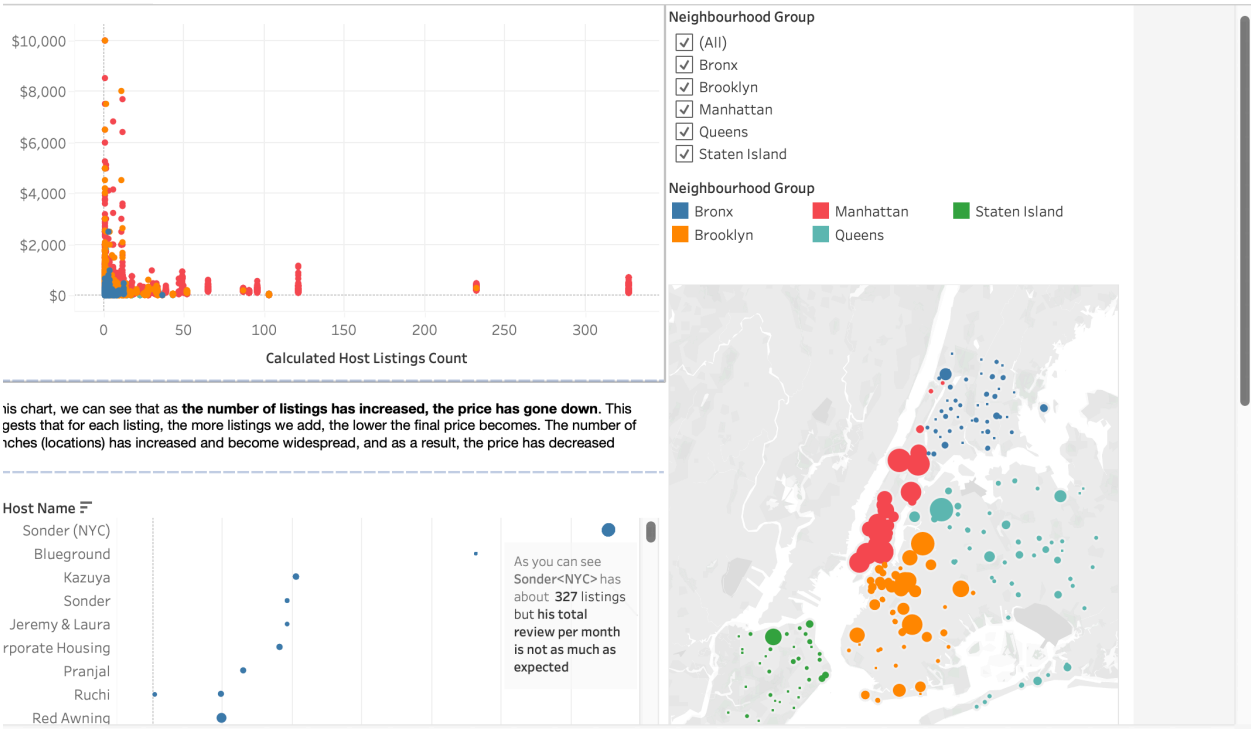

Pages
Columns
Calculated Host Listin..
Rows
Price (bin)
Filters
Marks
Automatic
Color
Size
Label
Detail
Tooltip
Shape
Reviews Per Month
Price (bin)
$0
$300
$600
$900
$1,200
$1,500
$1,800
$2,100
$2,400
$2,700
$3,000
$3,300
$3,600
$3,900
$4,200
$4,500
$4,800
0
50
100
150
200
250
300
Calculated Host Listings Count
Reviews Per Month
0.00
20.00

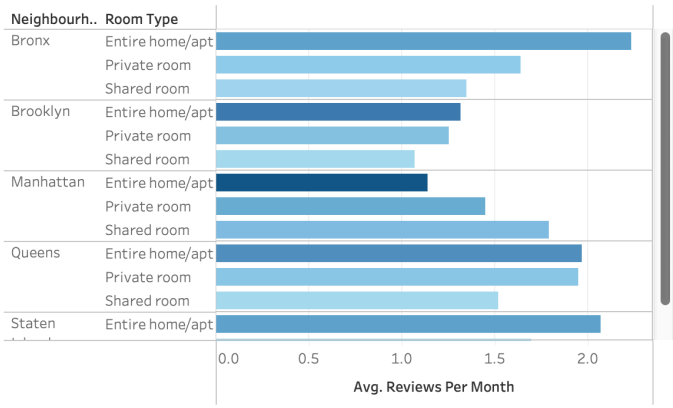

Pages
Columns
Longitude
Rows
Latitude
Latitude
Filters
Room Type
Marks
All
Automatic
Color
Size
Label
Detail
Tooltip
Multiple fields
Neighbourhoo..
Room Type
AVG(Latitude)
AVG(Latitude) (2)
Room Type
(All)
Entire home/apt
Private room
Shared room
Reviews Per Month
0.01
20.00
40.00
58.50
Calculated Host Listing...
1
100
200
327
© 2025 Mapbox © OpenStreetMap

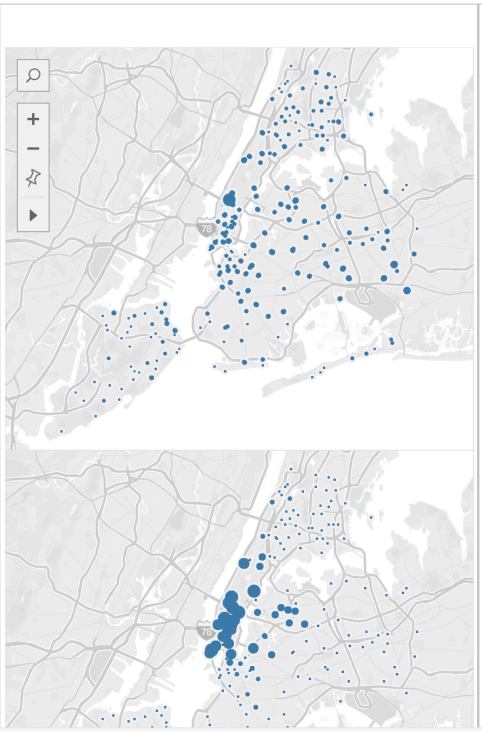

Pages
Columns
AVG(Reviews Per Mo..
Rows
Neighbourhood Group
Room Type
Filters
Room Type
Marks
Automatic
Color
Size
Label
Detail
Tooltip
MEDIAN(Price)
Neighbourh..
Room Type
Bronx
Brooklyn
Manhattan
Queens
Staten Island
Entire home/apt
Private room
Shared room
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
1.8
2.0
2.2
Avg. Reviews Per Month
Room Type
(All)
Entire home/apt
Private room
Shared room
MEDIAN(Price)
$30
$191



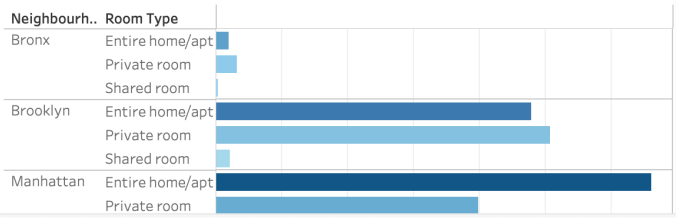
All dashboards:

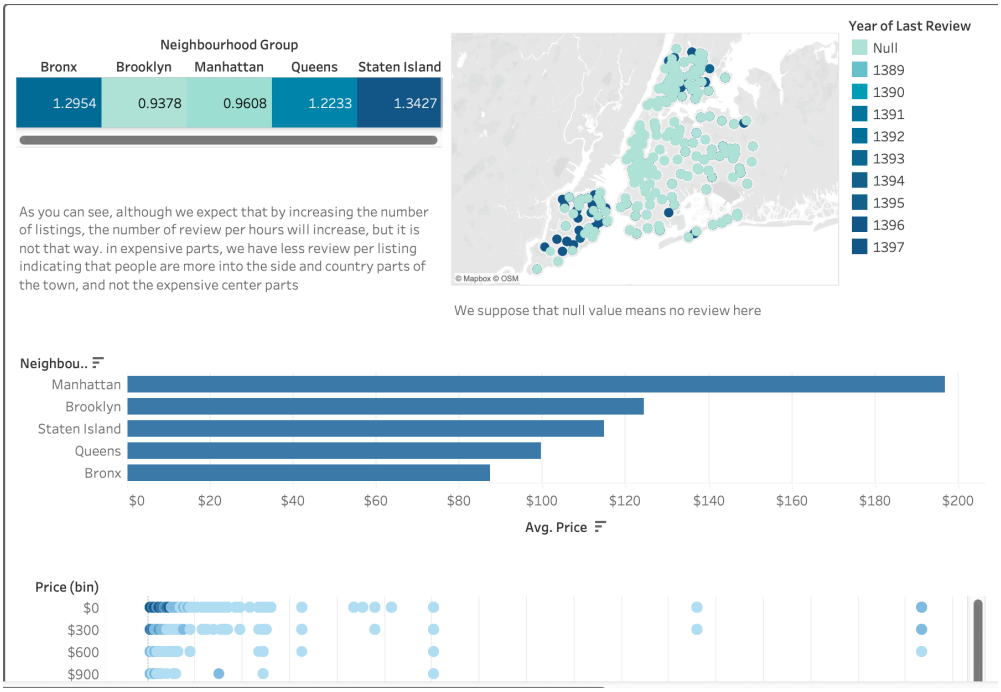






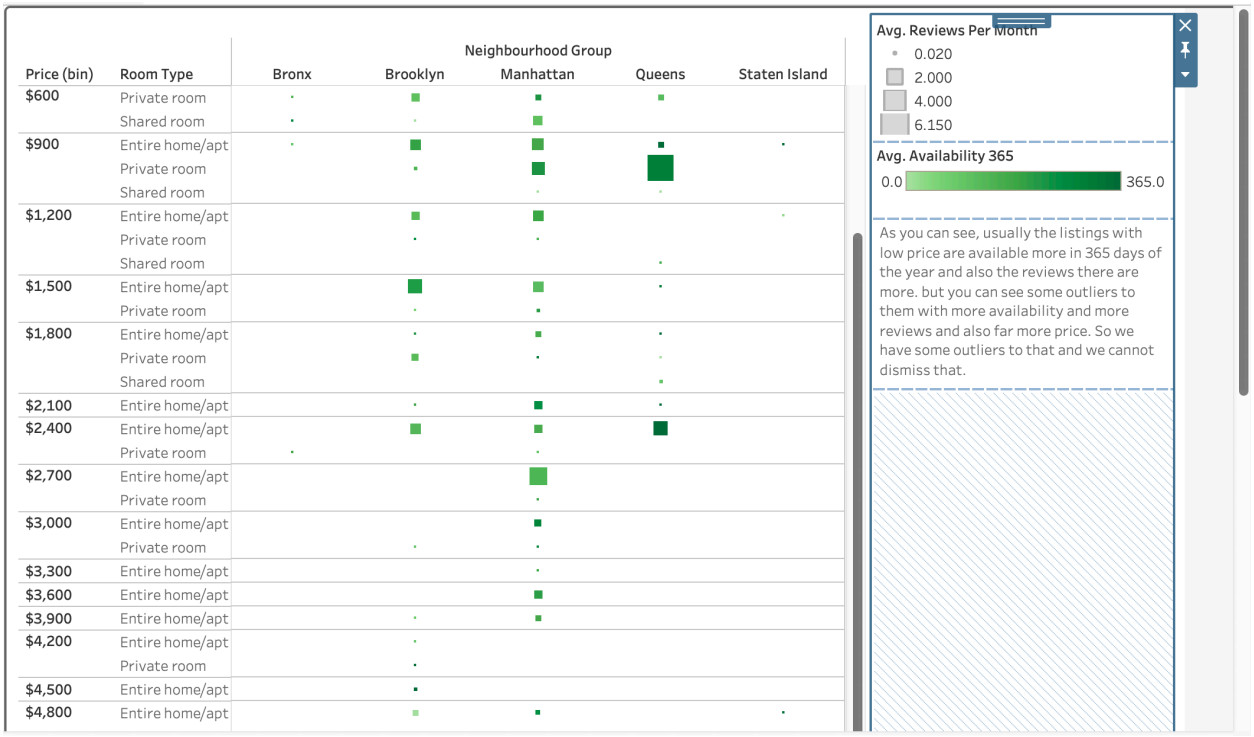
As you see, **Manhattan is a bias here**. in other neighborhood groups,**Usually entire homes are more hospitable than private room and private room are more hospitable than shared room**. but in Manhattan, due to the high price, this is vice versa.





As you can see, although we expect that by increasing the number of listings, the number of review per hours will increase, but it is not that way. in expensive parts, we have less review per listing indicating that people are more into the side and country parts of the town, and not the expensive center parts

We suppose that null value means no review here



Link

https://public.tableau.com/app/profile/alireza.karimi4809/viz/Book1_17439571260720/Story1