Synthetic Data Generation with Nonparametric Bayesian Methods

Highlight privacy-preserving data analysis

Giovanni Mele, Chiara Noemi Nesti, Chiara Tomasini,
Gianluca Villa, Andrea Violante, Stefano Zara

Supervised by:
Prof. Mario Beraha

Anno Accademico 2023/2024

Contents

1 Abstract

Nowadays, data play an increasingly central role in society; a substantial portion of these are sensitive data, and their direct use could lead to privacy issues. Nevertheless, utilizing these data is essential for gaining deep insights and making predictions about social and scientific phenomena. A commonly used solution in statistical applications is to generate synthetic data that preserve the properties of the original data while safeguarding privacy.

Our project is based on nonparametric Bayesian statistical (BNP) methods, such as data simulation from Polya Tree processes.

2 Theoretical Background[1]

2.1 ϵ-differential privacy

Differential Privacy (DP)[3,4] is a mathematically rigorous framework for releasing statistical information about datasets. It enables a data holder to share aggregate patterns of the group while protecting the privacy of individual data subjects. This is achieved by **injecting carefully calibrated noise** into statistical computations such that the utility of the statistics is preserved while limiting what can be inferred about any individual in the dataset.

Differentially private algorithms are widely used. For example, some government agencies utilize them to publish demographic information or other statistical aggregates while ensuring the confidentiality of survey responses. Similarly, companies leverage these algorithms to collect information about user behavior while controlling what is revealed about individuals.

An algorithm is said to be differentially private if an observer, given its output, cannot determine whether a particular individual's information was used in the computation.

In 2006, Dwork, McSherry, Nissim, and Smith introduced the concept of **ϵ-differential privacy**, a mathematical definition of the privacy loss associated with any data release drawn from a statistical dataset. This definition requires that a change to a single entry in a database causes only a small change in the probability distribution of the outputs of the measurements.

Definition. Let ϵ be a positive real number and $\mathcal{A}$ be a randomized algorithm that takes a dataset as input. The algorithm $\mathcal{A}$ said to provide ϵ-differential privacy if for all datasets $\mathcal{D}_1$ and $\mathcal{D}_2$ that differ on a single element (i.e. the data of one person), and all subsets $\mathcal{S}$ of image of $\mathcal{A}$:

$$\frac{\mathcal{P}(\mathcal{A}(\mathcal{D}_1) \in \mathcal{S})}{\mathcal{P}(\mathcal{A}(\mathcal{D}_2) \in \mathcal{S})} \leq e^{\epsilon}$$

2.2 Dirichlet process

Nonparametric Bayesian models (BNP) are widely used in statistics for estimating the density of an unknown distribution G. A distinctive feature of the Dirichlet Process (DP) is its capability to represent G as a weighted combination of point masses.

A DP with parameters M and G_0 is defined on a probability space S. In this framework, M is the concentration parameter, while G_0 serves as the base measure, acting as the "center" of the distribution generated by the process.
Respect to the parametric setting where the Dirichlet process (DP) is based on the inference about an

unknown distribution G on the basis of an observed i.i.d. sample

$$y_i \mid G \sim \text{iid } G, \quad i = 1, \ldots, n$$

If we wish to proceed with Bayesian inference, we need to complete the model with a prior probability model for the unknown parameter G. Assuming a prior model on G requires the specification of a probability model for an infinite-dimensional parameter, that is, a Bayesian Non Parametric prior.

Definition (Dirichlet process): Let $M > 0$ and G_0 be a probability measure defined on S. A DP with parameters (M, G_0) is a random probability measure G defined on S which assigns probability $G(B)$ to every (measurable) set B such that for each (measurable) finite partition $\{B_1, \ldots, B_k\}$ of S, the joint distribution of the vector $(G(B_1), \ldots, G(B_k))$ is the Dirichlet distribution with parameters $(MG_0(B_1), \ldots, MG_0(B_k))$.

The distribution can be expressed by

$$G \mid M, G_0 \sim \text{DP}(MG_0)$$

One important property of the DP is its large support, which allows G to approximate any probability measure with the same support as G_0. The proximity of G to G_0 is regulated by the parameter M; as M increases, G becomes increasingly concentrated around G_0.

2.3 Polya Tree

Recall that Dirichlet Process (DP) random probability measures G are inherently discrete. An elegant alternative that overcomes this limitation is the Polya Tree (PT) prior, which actually includes DP models as a special case. Unlike the DP, however, the PT prior allows for generating continuous distributions with probability one through an appropriate choice of parameters. Conceptually, the PT defines a random histogram.

Consider a histogram with bins defined by a partition of the sample space into non-empty subsets, denoted as $\{B_l, l = 0, \ldots, 2^m - 1\}$ (where 2^m represents the partition size, in anticipation of the discussion that follows). For each bin B_l, we can define random probabilities $G(B_l)$.

Now, refine this histogram by splitting each bin into two sub-bins: $B_l = B_{l0} \cup B_{l1}$. Then, define random probabilities for the refined histogram by setting $Y_{l0} = G(B_{l0} \mid B_l)$ and $Y_{l1} = 1 - Y_{l0} = G(B_{l1} \mid B_l)$. The recursive refinement of bins creates a sequence of nested partitions, which is the fundamental concept behind the PT prior.

As shown in Figure 1, the process begins with a partition of the interval $[0, 1]$, which is progressively subdivided into smaller intervals. At each level, a random Beta distribution determines how each bin is split, and the recursion continues with each new sub-bin being partitioned further.
The recursive structure is captured by the sequence of binary indices $\epsilon = \epsilon_1 \cdots \epsilon_m$, which represent paths through the tree. The probability associated with a given path is the product of the conditional probabilities for each subinterval, as indicated by the Beta random variables at each level. This recursive partitioning results in a sequence of nested intervals that define the structure of the Polya Tree.

The PT prior is the random probability measure G that arises when the Y_{ϵ_0}'s are independent beta random variables.

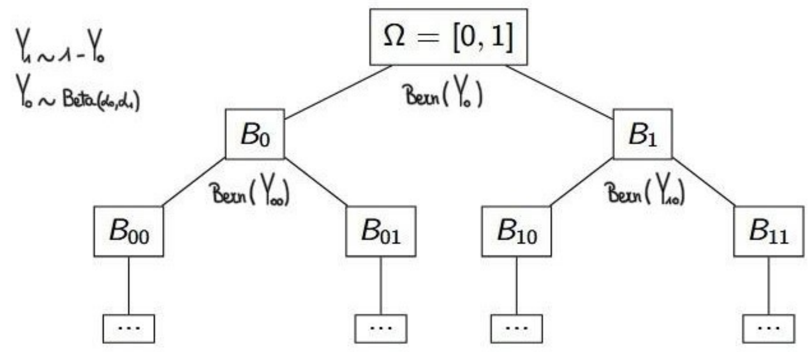


Figure 1: Illustration of the recursive partitioning process in the Polya Tree. Each bin B_l is refined by dividing it into two sub-bins, and random probabilities are assigned to each sub-interval.

$\Omega = B_0 \cup B_1$. Thus the subindices of the partitioning subsets are sequences $\epsilon = \epsilon_1 \cdots \epsilon_m$ of binary indicators $\epsilon_j \in \{0, 1\}$ and $G(B_{\epsilon_1 \cdots \epsilon_m}) = \prod_{j=1}^{m} Y_{\epsilon_1 \cdots \epsilon_j}$. The PT prior is the random probability measure G that arises when the Y_{ϵ_0}'s are independent beta random variables.

Definition. Let $\{B_\epsilon\}$, identified with Π, be a sequence of nested binary partitions as described before, and let $A = \{\alpha_\epsilon \mid \epsilon \in E\}$ be a collection of nonnegative numbers. A random probability measure G on Ω is said to be a Polya Tree (PT) with parameters $(\Pi\, A)$ if, for every $m = 1, 2, \ldots$ and every $\epsilon = \epsilon_1 \ldots \epsilon_m \in E_m$,

$$G(B_{\epsilon_1 \cdots \epsilon_m}) = \prod_{j=1}^{m} Y_{\epsilon_1 \cdots \epsilon_j},$$

where the conditional probabilities $Y_{\epsilon_1 \cdots \epsilon_{j-1} 0}$ are mutually independent beta random variables, given by

$$Y_{\epsilon_1 \cdots \epsilon_{j-1} 0} \sim \mathrm{Be}(\alpha_{\epsilon_1 \cdots \epsilon_{j-1} 0}, \alpha_{\epsilon_1 \cdots \epsilon_{j-1} 1}),$$

and $Y_{\epsilon_1 \cdots \epsilon_{j-1} 1} = 1 - Y_{\epsilon_1 \cdots \epsilon_{j-1} 0}$. We write $G \sim \mathrm{PT}(\{B_\epsilon\}, A)$.
In a Polya Tree (PT) model, prior centering ensures that the random probability measure G is aligned

with a desired prior measure G_0. The following are three commonly used methods for achieving prior centering:

1. **Prior Centering in PT($G_0, \mathcal{A}$):**
   This method ensures that the random probability measure G is centered at G_0 by aligning the moments of G with those of G_0. Specifically:

   $$E[G(B_{m,\epsilon})] = G_0(B_{m,\epsilon}),$$

   where $B_{m,\epsilon}$ are the intervals in the Polya Tree structure. The intervals and corresponding parameters $\alpha_{m,\epsilon}$ are chosen such that the expectation of G matches G_0 for each interval $B_{m,\epsilon}$. This guarantees consistency in centering the PT process.

2. **Prior Centering by Dyadic Quantiles:**
   An alternative approach involves constructing the intervals $B_{m,\epsilon}$ using dyadic quantiles of G_0. The intervals are defined as:

   $$B_{m,e} = \left[G_0^{-1}\left(\frac{\mathcal{N}(\epsilon)}{2^m}\right), G_0^{-1}\left(\frac{\mathcal{N}(\epsilon)+1}{2^m}\right) \right].$$

This ensures that G_0 divides the probability space equally among the partitions. The parameters $\alpha_{m,\epsilon}$ are then selected to ensure alignment with the expected probability mass $G_0(B_{m,\epsilon})$, maintaining the desired centering at G_0.

3. **Canonical Choices for $\alpha_{m,\epsilon}$:**
   A widely used method for prior centering involves specifying the parameters $\alpha_{m,\epsilon}$ in a structured form, such as:
   $$\alpha_{m,\epsilon} = c \cdot m^2, \quad \forall\epsilon,$$
   where c is a positive constant that controls the strength of the prior, and m^2 emphasizes the depth of the tree. This choice provides flexibility while balancing prior influence across levels of the tree. Other forms, such as $\alpha_{m,\epsilon} = c \cdot m^p$, have also been proposed, where p and c can be tuned to reflect varying levels of deviation from G_0.

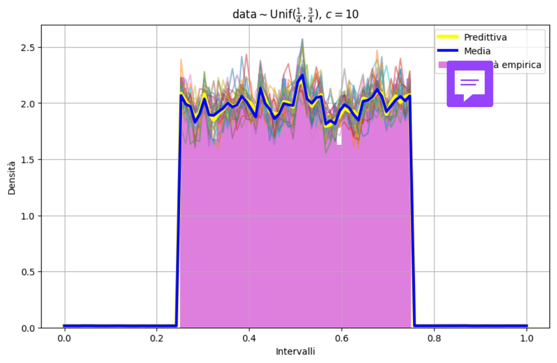
In our project, we opted for the canonical choice for $\alpha_{m,\epsilon}$, as it provides a structured and flexible framework for centering the Polya Tree. By experimenting with different values of the parameters c and p, we analyzed the trade-offs between prior strength and flexibility. This allowed us to better understand how these parameters influence the balance between prior knowledge and data-driven learning in the PT model.

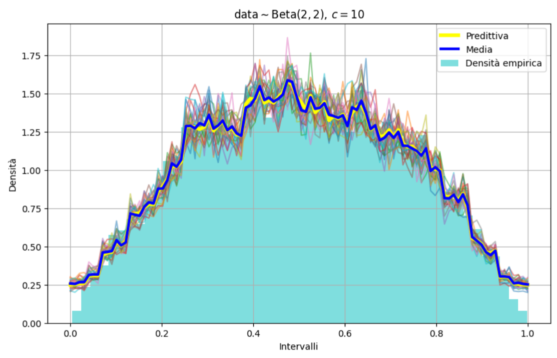
3 Practical Applications

3.1 Preliminary Analysis

[6] The first step was to investigate the properties of a Polya Tree and how it varies with changes in its parameters.

Since we initially did not have a dataset, we generated data from a known distribution within $[0, 1]$, such as a $\mathcal{B}eta$ or a $\mathcal{U}niform$ distribution, and used the latter for most of our analyses. We set our prior as a Polya Tree with depth $m = 10$ (with initial parameters α all set to $c \cdot m^2$, where c is fixed). Since the Polya Tree is a conjugate model, we updated the parameters of our Polya Tree using the generated data.
Next, we performed **sampling from the posterior**, extracting 30 distributions, and repeating the process both with initial data sampled from a $\mathcal{U}(\frac{1}{4}, \frac{3}{4})$ and from a $\mathcal{B}eta(2, 2)$. Here we present the samples for $c = 10$, where the mean of the distributions is highlighted in blue.





The code begins by defining a set of parameters and iterating over a list of values for c, which is a parameter used in the Polya Tree model to influence the strength of the prior. In this case, $c = 10$, and the code runs for a total of 30 samples. The depth of the Polya Tree is set to 10, meaning the tree has 10 levels of branching.

For each value of c, the function `polya_tree_posterior` is called to compute the posterior distribution of the Polya Tree. This function returns two arrays: all_alpha, which represents the posterior parameters at each level of the tree, and all_B, which contains the intervals at each level. The final intervals at the deepest level (all_B[-1]) are extracted and stored in `final_intervals`.
The next step involves sampling from the Polya Tree. This is done by calling the function `sample_from_polya_tree` repeatedly. For each sample, the function generates a distribution, which is then added to a list of sampled distributions. Additionally, the mean of all the sampled distributions is calculated by accumulating them and averaging at the end of the loop.
Once the sampling process is complete, the code calculates the midpoints of the intervals in `final_intervals` and their corresponding widths, which are used to normalize and plot the distributions. A plot is generated for each sampled distribution, where the area under each curve is normalized so that the total area sums to 1. These distributions are then interpolated for smooth plotting.

The **predictive distribution** is calculated by calling the `compute_predictive` function, which takes the parameters all_alpha and all_B from the Polya Tree. This predictive distribution is then normalized and plotted alongside the mean distribution of the samples.
Finally, the code visualizes all the results: it plots the empirical density from the data, the predictive distribution calculated by the Polya Tree model, and the mean of the sampled distributions. The plot also includes a histogram of the sample data and labels for clarity. The plot is displayed using Matplotlib, with labels, a title, and a grid for better readability.

```python
c_values = [10]

num_samples = 30
depth = 10

for c in c_values:
    all_alpha, all_B = polya_tree_posterior(data=sample, c=c, depth=depth)
    final_intervals = all_B[-1]
    sampled_distributions = []
    mean = 0

    # Sampling from a Polya tree
    for _ in range(num_samples):
        _, sampled_distribution = sample_from_polya_tree(all_alpha)
        sampled_distributions.append(sampled_distribution)
        mean += sampled_distribution

    mean /= num_samples
    x_points = (final_intervals[:, 0] + final_intervals[:, 1]) / 2
    widths = np.diff(final_intervals, axis=1).flatten()

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

    for sampled_distribution in sampled_distributions:
        area = np.sum(sampled_distribution * widths)
        normalized_distribution = sampled_distribution / area
        f = interp1d(x_points, normalized_distribution, kind='linear',
            fill_value="extrapolate")
        x_interp = np.linspace(final_intervals[0, 0], final_intervals[-1, 1],
            100)
        y_interp = f(x_interp)
        plt.plot(x_interp, y_interp, alpha=0.5)

    sampled_distribution = compute_predictive(all_alpha, all_B)
    # Plot of the predictive
    area = np.sum(sampled_distribution * widths)
    normalized_distribution = sampled_distribution / area
    f = interp1d(x_points, normalized_distribution, kind='linear', fill_value=
        "extrapolate")
    x_interp = np.linspace(final_intervals[0, 0], final_intervals[-1, 1], 100)
    y_interp = f(x_interp)
    plt.plot(x_interp, y_interp, alpha=1, lw=4, color="yellow", label="
        Predittiva")

    # Plot of the mean
    sampled_distribution = mean
    area = np.sum(sampled_distribution * widths)
    normalized_distribution = sampled_distribution / area
    f = interp1d(x_points, normalized_distribution, kind='linear', fill_value=
        "extrapolate")
    x_interp = np.linspace(final_intervals[0, 0], final_intervals[-1, 1], 100)
    y_interp = f(x_interp)
    plt.plot(x_interp, y_interp, lw=3, alpha=1, color="blue", label="Media")

    plt.hist(sample, color='c', label='Empiric density', bins=50, density=True
        , alpha=0.5)

    plt.xlabel("Intervals")
    plt.ylabel("Density")
    plt.legend()

    plt.title(r"$\text{data}\sim\text{Beta}(2, 2), \, c = {" + str(c) + "}$")
    plt.grid()
    plt.show()
```

We can also compute the predictive distribution. Its theoretical formulation is the following:

$$P\left(Y_{n+1} \in B_\epsilon \mid y_1, \ldots, y_n\right) = \prod_{m=1}^{M} \frac{\alpha^{*(n+1)}_{\epsilon_m(Y_{n+1})}}{\alpha^{*(n+1)}_{\epsilon_{m-1}(Y_{n+1})0} + \alpha^{*(n+1)}_{\epsilon_{m-1}(Y_{n+1})1}}$$

It's the product of the expected values of the $\mathcal{B}eta$ random variables at the m-levels.

This expression reads that the probability for a new observation to fall into the subinterval indexed by binary path ϵ, given the previous observations, is the product, ranging from level 1 to level M, of these ratios. These ratios are, in fact, nothing more than the expected values of the $\mathcal{B}eta$ random variables that we have at each sublevel m. Specifically, in the ratios, the numerator corresponds to the α value associated with the path that the new observation is currently following at level m, while the denominator corresponds to the sum of the α values for the two paths that the observation could have taken at level $m-1$. The asterisks at the top indicate that the α values have already been updated based on the previous observations.

In the previous graphs, the predictive distribution is highlighted in yellow.

The `compute_predictive` function is designed to compute the predictive probabilities for a Polya Tree model, which is used for probabilistic inference. This function takes two main inputs: `all_alpha` and all_B, which define the structure and parameters of the tree.

First, the function determines the depth of the tree by examining the length of all_B. The depth dictates how many levels the tree has. Then, for each possible path in the tree, the function computes the probability by iterating over the levels of the tree. At each level, the function uses the all_alpha values to determine the branching probabilities.
The key logic involves traversing the tree from the root to the leaves. For each path, the function calculates the probability of following that path by comparing the α values associated with the nodes in the tree. The probabilities are accumulated as the function moves through the tree, considering the branching structure defined by all_B.
Finally, the function returns an array, predictive_prob, which contains the calculated probabilities for each path at the deepest level of the tree. These probabilities represent the predictive distribution based on the input data.

```python
def compute_predictive(all_alpha, all_B):
    depth = len(all_B) - 1
    k = 2 ** depth
    predictive_prob = np.zeros(k)
    for i in range(k):
        prob = 1
        path = idx2path(i, len_output_path=depth)
        for m in range(1, depth+1):
            index = path2idx(path[:m])
            if index % 2 == 0:
                prob = prob * all_alpha[m][index] / (all_alpha[m][index] +
                    all_alpha[m][index+1])
            else:
                prob = prob * all_alpha[m][index] / (all_alpha[m][index] +
                    all_alpha[m][index-1])

        predictive_prob[i] = prob
    return predictive_prob
```

3.2 Exploring when c varing

Both the predictive and the posterior distributions generated by a PT process depend strongly on the parameter c. Recall that we have defined $\alpha = c \cdot m^2$ at the last level M. Let us explore how these distributions vary with changes in the parameter c, considering the following scenarios:

Case 1: $c = 1$

When $c = 1$, we set

$$\alpha = c \cdot m^2 = 1 \cdot 10^2 = 100$$

Here, α is significantly smaller compared to the number of observations (10.000) used to update the parameters. As a result:

- The data strongly updates the prior α values.
- The posterior distribution closely follows the empirical distribution of the observations, deviating significantly from the prior.

Case 2: $c = 1.000$

For $c = 1.000$, we have

$$\alpha = c \cdot m^2 = 1.000 \cdot 10^2 = 100.000$$

In this scenario:

- α is much larger than the number of observations (10.000).
- The posterior distribution is influenced more by the prior rather than the empirical distribution, as the observations carry less weight in updating the parameters.

Case 3: $c \to \infty$

As c approaches infinity, α becomes arbitrarily large:

$$\alpha = c \cdot m^2 \to \infty.$$

In this limiting case:

- The influence of the data on the posterior distribution vanishes entirely.
- The posterior distribution converges to the prior distribution, as the observations lose their power to update α.

This analysis highlights how the parameter c determines the balance between the influence of the prior and the empirical data on the posterior distribution.

To better understand the effect of varying the parameter c, we present six plots. The first row corresponds to the posterior distributions derived from data sampled from the uniform distribution $\mathcal{U}niform(\frac{1}{4}, \frac{3}{4})$ for different values of c. The second row corresponds to the posterior distributions derived from data sampled from the $\mathcal{B}eta(2, 2)$ distribution. Each plot illustrates how the posterior changes as c increases, demonstrating the balance between the influence of the empirical data and the prior.

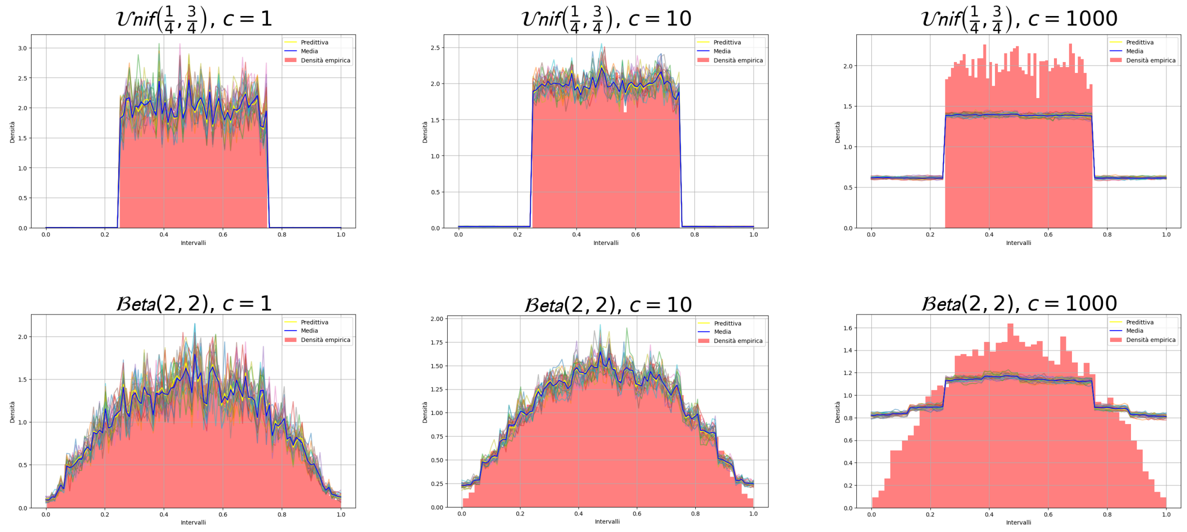


Figure 2: Comparison of posterior distributions for varying c values under two prior distributions: $\mathcal{U}(1/4, 3/4)$ (top row) and $\mathcal{B}eta(2, 2)$ (bottom row).

3.3 Measures and metrics

To quantify the distance between the predicted posterior distribution and the empirical distribution derived from the dataset, we focused on specific measures and metrics. Initially, we performed diagnostics for the parameter c by analyzing two metrics:

- **L_1 Distance:**[5] Measures the absolute difference between two distributions. It provides a straightforward way to assess how much the predicted and empirical distributions diverge overall:

  $$L1(P, Q) = \int |P(x) - Q(x)| \mathrm{d}x$$

  where $P(x)$ and $Q(x)$ are the posterior and empirical distributions, respectively.

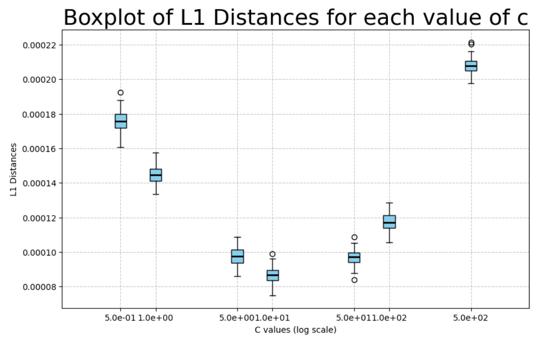
- **Hellinger Distance:** Quantifies the similarity between two probability distributions. It is particularly sensitive to differences in the shape and overlap of the distributions:

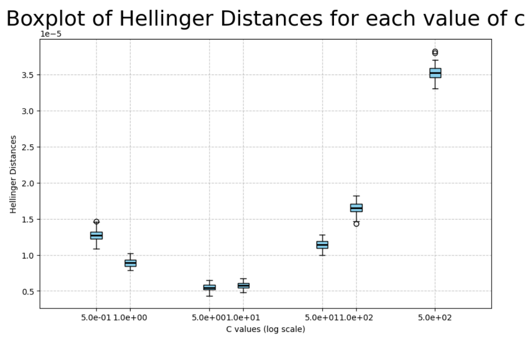
  $$H(P, Q) = \sqrt{\frac{1}{2} \int \left(\sqrt{P(x)} - \sqrt{Q(x)} \right)^2 \mathrm{d}x}.$$

  where $P(x)$ and $Q(x)$ are the posterior and empirical distributions, respectively.

We computed these distances between the posterior distribution and the empirical distribution for various values of the parameter c. The results revealed that the value of c that minimizes both metrics lies in the middle of the tested range. This behavior suggests a balance between the influence of the empirical data and the prior.

The figure below illustrates the computed L_1 and Hellinger distances for varying values of c. As shown, the minimum distance is observed at an intermediate value of c, indicating the optimal trade-off between data and prior.





In the first case, let $c = 5$. In fact, if a smaller value of c is chosen, some distance would appear between the posterior and the predictive distributions, caused by the variability in the data. On the other hand, if larger values of c are chosen, the posterior distribution no longer follows the empirical distribution, causing the metrics to explode to infinity.

If we select a smaller value of c, we introduce some discrepancy between the posterior and the predictive due to the inherent variability in the data. For large values of c, the posterior distribution deviates significantly from the empirical distribution, leading to infinite values in the metrics.

In other words, the relationship between the posterior and predictive distributions becomes unstable as c increases, as shown by the explosion of the metrics.

This code computes and visualizes the L1 and Hellinger distances between samples generated from a Polya Tree posterior distribution and a reference distribution. The key steps include sampling from the Polya Tree, calculating the distances for each set of samples, and then plotting a boxplot to compare the distributions of these distances for different values of the hyperparameter c.

The hyperparameter c is varied over a set of predefined values, and for each value of c, the code calculates the L1 and Hellinger distances between the generated samples and the reference. These distances are stored in two lists, `L1_distances` and `Hellinger_distances`, which are later used to create boxplots.

```python
c_values = [0.5, 1, 5, 10, 50, 100, 500]

num_samples = 100

L1_distances = []
Hellinger_distances = []
for c in c_values:
    all_alpha, all_B = polya_tree_posterior(data=sample, c=c, depth=depth)
    final_intervals = all_B[-1]
    L1_distance = []
    Hellinger_distance = []

    for _ in range(num_samples):
        _, g = sample_from_polya_tree(all_alpha)
        L1_distance.append(np.sum(np.abs(g - f))/k)
        Hellinger_distance.append(np.sum(((np.sqrt(g) - np.sqrt(f))**2))/k)

    L1_distances.append(L1_distance)
    Hellinger_distances.append(Hellinger_distance)

plt.figure(figsize=(10, 6))          # Change the commented part to change the
    metric
box = plt.boxplot(
    L1_distances,
    # Hellinger_distances,
    positions=np.log10(c_values),
    widths=0.1,
    patch_artist=True)

uniform_color = 'skyblue'
for patch in box['boxes']:
    patch.set_facecolor(uniform_color)
    patch.set_edgecolor('black')

for median in box['medians']:
    median.set_color('black')
    median.set_linewidth(2)

plt.xscale('linear')
plt.xticks(np.log10(c_values), labels=[f"{c}" for c in c_values])
plt.xlabel('C values (log scale)')

plt.ylabel('L1 Distances')
plt.title('Boxplot of L1 Distances for each value of c')
```

```

# plt.ylabel('Hellinger Distances')
# plt.title('Boxplot of Hellinger Distances for each value of c')

plt.grid(True, linestyle='--', alpha=0.7)

plt.show()
```

The metrics used thus far, such as L_1 and Wasserstein distances, illustrate the relationship between theoretical and simulated data. This comparison is feasible because the distribution of the original data is known, as the data were drawn from uniform or Beta distributions.
However, it is necessary to have a distance metric that can relate the original data to the simulated data when the distribution is unknown.

In this more general context, we employ the **Wasserstein distance**[7].
The **Wasserstein distance** is a metric used to quantify the difference between two probability distributions. It is rooted in the theory of optimal transport, which involves finding the most efficient way to transform one distribution into another, minimizing the associated cost. In its general form, the Wasserstein distance of order p between two probability measures μ and ν on a metric space (X, d) is given by:

$$W_p(\mu, \nu) = \left(\inf_{\gamma \in \Pi(\mu,\nu)} \int_{X \times X} d(x, y)^p \, d\gamma(x, y) \right)^{1/p}$$

where $\Pi(\mu, \nu)$ denotes the set of all couplings (joint distributions) between μ and ν, and $d(x, y)$ is the cost function (often the distance between points x and y in the metric space). The term p defines the order of the distance, with $p = 1$ corresponding to the classic **Wasserstein-1** distance and $p = 2$ corresponding to the **Wasserstein-2** distance.

The Wasserstein distance has several important properties: it is a true metric, it satisfies the triangle inequality, and it is sensitive to the geometry of the underlying space. This makes it particularly useful for comparing distributions in situations where the structure of the data is important, such as in machine learning tasks involving high-dimensional data or complex distributions.

This distance is applicable to general spaces and allows for comparisons between atomic measures and point densities. It can be explicitly computed using the Python package **POT** (Python Optimal Transport). We use the Wasserstein-2. Our implementation is provided below.

```
import ot

def compute_empirical_density(vector):
  unique, counts = np.unique(vector, return_counts=True)
  frequencies = counts / len(vector)
  density_vector = np.array([frequencies[np.where(unique == val)[0][0]] for
      val in vector])
  density_vector = density_vector/sum(density_vector)
  return density_vector

n_list = 10 ** np.linspace(1,4,15) # n = 10 ^ {1...3}
wasserstein_distances = []

f = 1/2 # m = n * f
n_points_per_boxplot = 15

c = 10 # parametri polya tree
```

```
depth = 10

for n in n_list:
  n = int(n)
  m = int(n * f)
  wasserstein_distances_n = []

  for k in range(n_points_per_boxplot):

      # original_sample = sample_beta(2,2,size = n)
      original_sample = np.random.uniform(1/4, 3/4, size = n)

      all_alpha, all_B = polya_tree_posterior(data=original_sample, c=c, depth
          =depth)
      intervals = all_B[-1]
      posterior = compute_predictive(all_alpha, all_B)
      pt_sample = sample_from_categorical_distribution(intervals =  intervals,
           probabilities = posterior, n = m)
      original_data_empirical_density = compute_empirical_density(
          original_sample)
      pt_data_empirical_density = compute_empirical_density(pt_sample)

      M = ot.dist(original_sample.reshape(-1,1), pt_sample.reshape(-1,1),
          metric='euclidean')
      Wd = ot.emd2(original_data_empirical_density, pt_data_empirical_density,
           M)
      wasserstein_distances_n.append(Wd)
  wasserstein_distances.append(wasserstein_distances_n)
```

In our case, we set $m = n/2$, and as shown in the following boxplots, the variability decreases significantly as n increases.

A smaller Wasserstein distance indicates better quality of the synthetic data compared to the original. Indeed, the Wasserstein distance decreases as n increases, as the data become more stable and less susceptible to random variation.

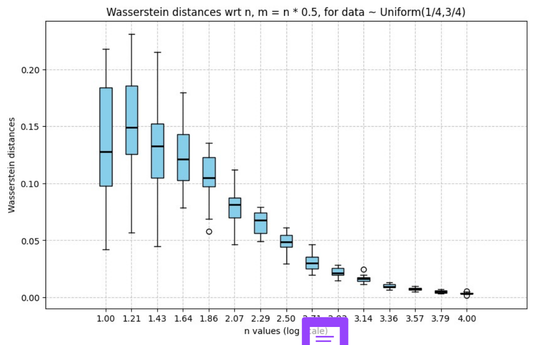


Figure 3: Boxplot showing the variability of the data for a $Uniform(\frac{1}{4}, \frac{3}{4})$.

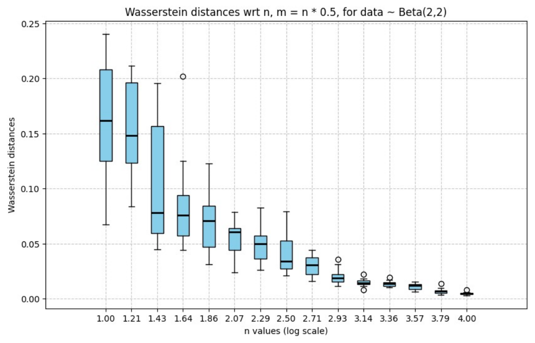


Figure 4: Boxplot showing the variability of the data for a $\mathcal{B}eta(2, 2)$.

3.4 Analysis of Parameter c and its impact on ϵ-Differential Privacy

In theoretical terms, the concept of ϵ-*differential privacy* ensures that a mechanism protects data privacy by limiting the influence of any single data point on the output. We now consider the specific case of our mechanism, parameterized by c, analyzing its impact on differential privacy.

Choosing a small c is beneficial for achieving a high level of privacy, as the data are strongly modified according to the posterior distribution of the PT process. However, we observe that the parameter ϵ, which quantifies differential privacy, tends to smaller values (i.e., privacy improves) as c increases. When c is large, the initial dataset carries less weight in the posterior updating of the PT. In this scenario, a potential adversary, who knows the initial dataset (except for one value) and the mechanism used to generate new data, faces greater difficulty in identifying the missing value by examining the likelihood of a set of generated datasets.

Consider a simplified case where all generated datasets have m identical observations. In this situation:

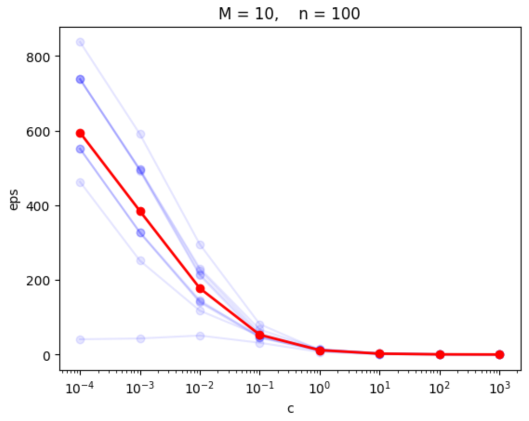


Figure 5: The supremum of the density ratios decreases as c increases

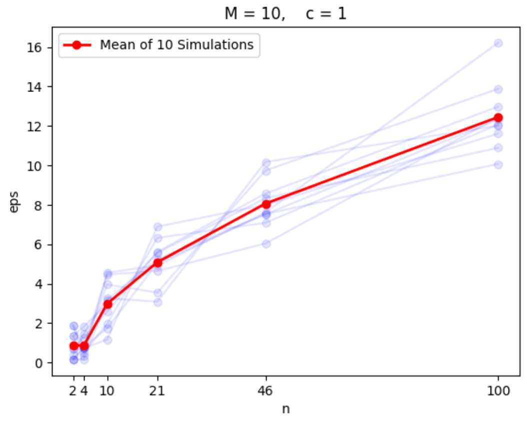


Figure 6: The supremum of the density ratios increases as the initial sample size grows

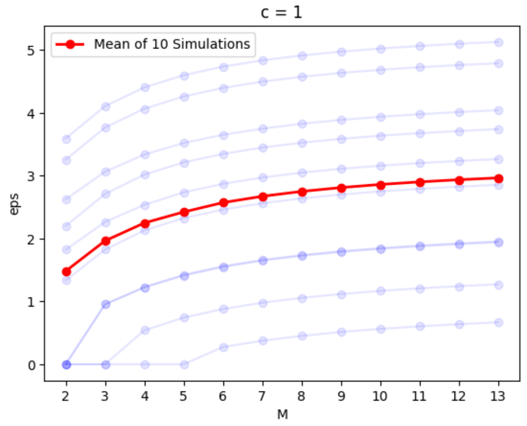


Figure 7: The supremum of the density ratios increases as the depth of the PT process grows

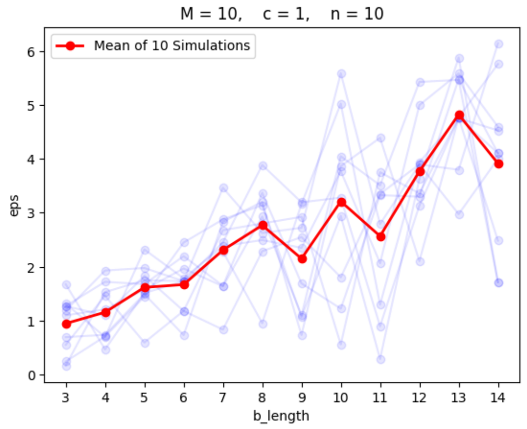


Figure 8: The supremum of the density ratios increases as the size of the released dataset grows

In the general case (with a Hamming distance of 1 between datasets x and y), we attempted to compute the supremum using both random sampling and uniform coverage of the dataset domain. Here, it is also evident that ϵ-differential privacy is better satisfied as c increases.
Checking for ϵ-differential privacy remains computationally demanding. To address this complexity, we analyzed a relaxation of it, namely ϵ-δ *differential privacy.*

Definition.[2] A mechanism satisfies (ϵ, δ) probabilistic differential privacy is, given $q_n(z|\mathbf{x})$ the density of $Q_n(\cdot|\mathbf{X} = \mathbf{x})$, there exists a $\delta > 0$ such that $Q_n(A_{\mathbf{x},\mathbf{y}}|\mathbf{X} = \mathbf{x}) > 1 - \delta$ for all $\mathbf{x},\mathbf{y} : h(\mathbf{x},\mathbf{y}) = 1$ where $A_{\mathbf{x},\mathbf{y}} := z \in \mathcal{X}^m : q_n(z|\mathbf{x}) \leq e^{\epsilon} \cdot q_n(z|\mathbf{y})$.

```python
def bnp_mechanism(X, m, theta, sigma, H = lambda: np.random.uniform(0, 1/4),
    show=False):

  k = len(np.unique(X))
  n_i = np.array([list(X).count(x) for x in np.unique(X)])
  n = len(X)
  Z = []

  for j in range(m):
    probs = np.append((n_i - sigma)/(theta + n),(theta + sigma*k)/(theta + n))
    if show: print(probs)
    choice_index = np.random.choice(k + 1, p=probs)
    if show: print(choice_index)

    if choice_index == k:
      Z_j = H()
      n_i = np.append(n_i, 1)
      k += 1
      X = np.append(X, Z_j)
    else:
      Z_j = np.unique(X)[choice_index]
      n_i[choice_index] += 1

    n += 1
    Z.append(Z_j)

  return Z
```

ϵ-δ *differential privacy* is a weaker property than pure ϵ- *differential privacy.* However, as described above, it allows for achieving analogous results without requiring excessively computationally expensive procedures. Specifically, if we set $\delta = 0$, we would recover exact epsilon-differential privacy. However, allowing δ to be a small positive number between 0 and 1 provides greater flexibility, as δ can be interpreted as the fraction of dataset combinations that do not satisfy epsilon-differential privacy for a fixed ϵ.

To estimate this fraction, we can run a Monte Carlo simulation on a set of density ratios, fix an ϵ, and then draw a threshold line at $y = e^{\epsilon}$. The fraction of points falling above this line gives us the value of δ. As shown in Figure 9

This approach is inspired by the work of Lazzarini[2]. The algorithm proceeds as follows: we first generate a dataset X randomly from the sample distribution, then generate a new dataset Y by randomly changing one element of X (maintaining a Hamming distance of 1), and replacing the changed element with a new entry sampled from the same distribution. Finally, we generate dataset Z using a mechanism known as the "BNP mechanism."

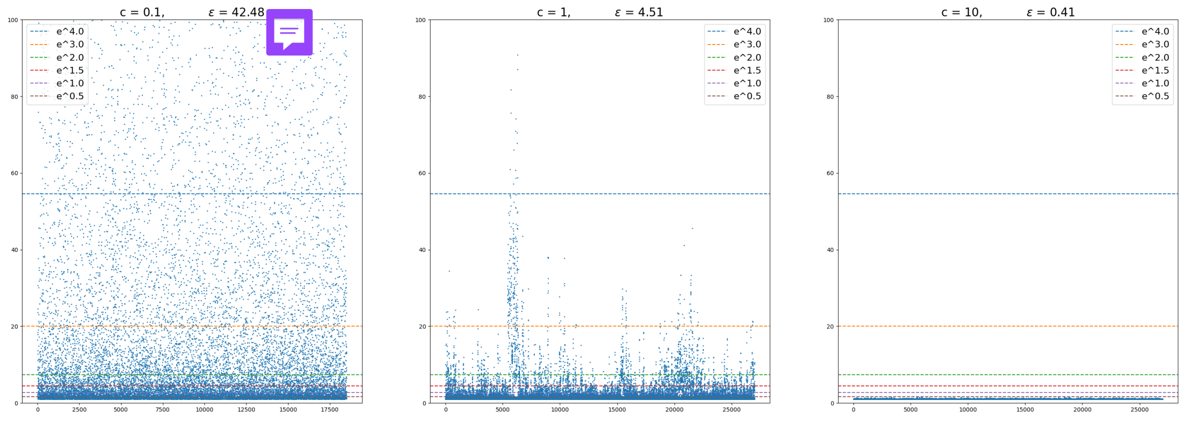


Figure 9: Monte Carlo simulation on a set of density ratios, fix an ϵ, and then draw a threshold line at $y = e^{\epsilon}$

The BNP mechanism allows us to iteratively sample a new entry for Z from a categorical distribution with as many possible outcomes as the number of unique values in X, plus one additional outcome.

Regarding the generation of Y, we implemented two different methods. The first follows the procedure outlined in the thesis, which involves selecting one of three possible ways to change and replace an element of X based on the outcome of a random uniform variable. The second method is more intuitive, where we simply replace an element of X by sampling a new entry from the same sample distribution. Both methods yielded similar approximations of the real supremum and confirmed that by increasing c, ϵ decreases.

4 Real applications

Esempio con dataset vero

5 Conclusions

Under the framework of Differential Privacy, we examined the assumptions necessary to ensure privacy guarantees for our proposed method.

It is worth noting that many mechanisms in the literature allow for tuning parameters to control the desired level of privacy. In contrast, our approach leverages a nonparametric model, which is inherently more data-driven. Despite this, we demonstrated that privacy guarantees can still be maintained by appropriately regulating the amount of synthetic data released.
Statistically, our mechanism provides significant advantages in terms of the quality of the generated synthetic data. As the amount of synthetic and original data increases, the empirical distribution of the synthetic data gradually converges to the true data-generating distribution. Additionally, the mechanism facilitates straightforward computation of the one-step posterior predictive distribution based on the synthetic data, offering a practical advantage for data analysis.

References

[1] Peter Müller, Fernando Andrés Quintana, Alejandro Jara, Tim Hanson, *Bayesian Nonparametric Data Analysis*, Springer, 2015.

[2] Riccardo Lazzarini, *Bayesian Nonparametric Privacy-Preserving Synthetic Data Generation*, 2023.

[3] J. M. Abowd, *The U.S. Census Bureau Adopts Differential Privacy. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2018.

[4] Cynthia Dwork, Nitin Kohli, Deirdre Mulligan, *Differential privacy in practice: expose your epsilon!*, 2019

[5] Walter Rudin, *Functional Analysis*, 1991, McGraw-Hill Education

[6] Rachel Cummings, Damien Desfontaines, David Evans, Roxana Geambasu, Yangsibo Huang, Matthew Jagielski, Peter Kairouz, Gautam Kamath, Sewoong Oh, Olga Ohrimenko, Nicolas Papernot, Ryan Rogers, Milan Shen, Shuang Song, Weijie Su, Andreas Terzis, Abhradeep Thakurta, Sergei Vassilvitskii, Yu-Xiang Wang, Li Xiong, Sergey Yekhanin, Da Yu, Huanyu Zhang, Wanrong Zhang, *Advancing Differential Privacy: Where We Are Now and Future Directions for Real-World Deployment*, 14/04/2023, Harvard Data Science Review, https://hdsr.mitpress.mit.edu/pub/sl9we8gh

[7] Jian Shen, Yanru Qu, Weinan Zhang, Yong Yu, *Wasserstein Distance Guided Representation Learning for Domain Adaptation*, 05/07/2017, https://arxiv.org/abs/1707.01217