clt-zoo-exercise

February 5, 2026

1 CLT Zoo

This notebook presents an interactive, simulation-based illustration of the **Law of Large Numbers (LLN)** and the **Central Limit Theorem (CLT)**, two cornerstone results in probability theory with central importance in statistics and econometrics. The objective is not merely to restate these results abstractly, but to examine their **finite-sample behavior** under different distributional assumptions through controlled Monte Carlo experiments.

1.0.1 Setup;

Let $X_1, X_2, \ldots, X_n$ denote an independent and identically distributed sample drawn from a population with mean μ and variance σ^2. No normality assumption is imposed unless explicitly selected. The sample mean is defined as

$$\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i.$$

1.0.2 Law of Large Numbers (LLN);

Under standard regularity conditions (e.g. $E|X_1| < \infty$), the Law of Large Numbers implies that

$$\bar{X}_n \xrightarrow{p} \mu \quad \text{as } n \to \infty,$$

meaning that the sample mean converges in probability to the population mean. This result forms the theoretical backbone of a large class of econometric estimators, where sample averages serve as **consistent estimators** of population moments (Kolmogorov, 1933).

1.0.3 Central Limit Theorem (CLT);

Beyond consistency, the Central Limit Theorem characterizes the asymptotic distribution of the sample mean. In its classical Lindeberg–Lévy formulation, if the variance σ^2 is finite, then

$$\sqrt{n}(\bar{X}_n - \mu) \xrightarrow{d} \mathcal{N}(0, \sigma^2).$$

Equivalently,

$$\bar{X}_n \xrightarrow{d} \mathcal{N}\left(\mu, \frac{\sigma^2}{n}\right),$$

as $n \to \infty$ (Lindeberg, 1922; Lévy, 1937).

A commonly used standardized version is

$$Z_n = \frac{\sqrt{n}(\bar{X}_n - \mu)}{\sigma} \xrightarrow{d} \mathcal{N}(0, 1),$$

which provides the basis for large-sample inference in applied econometrics (Gujarati, 2011).

1.0.4 Role of distributional assumptions;

An important point emphasized in the classical literature is that the CLT does **not** require the underlying population to be normally distributed. However, the classical Gaussian limit relies critically on the existence of **finite second moments**. When this condition fails—most notably for **heavy-tailed distributions** with infinite variance—the standard normal approximation may no longer apply. Convergence can be substantially slower or may occur toward a non-Gaussian limit (Feller, 1968; Billingsley, 1995).

1.0.5 Purpose of this notebook;

This notebook makes these theoretical results operational. By allowing the user to vary the underlying distribution, its parameters, and the sample size, the simulations illustrate:

- how the LLN manifests in finite samples,
- how quickly the CLT approximation becomes accurate,
- and under which distributional conditions the classical results break down.

The emphasis is on **visual intuition** and **finite-sample behavior**, rather than purely asymptotic statements.

1.0.6 Contact;

This notebook was prepared as part of an exploratory and pedagogical project on probability theory and econometrics.
For questions, suggestions, or comments regarding the implementation or interpretation of the simulations, please feel free to get in touch.

Author: *[Muhammed İkbal Yılmaz]*
Email: *[myucanlar@gmail.com]*

[8]:
```
import os
from pathlib import Path
import numpy as np
import matplotlib.pyplot as plt
from math import sqrt, pi, gamma, log
from IPython.display import display
import ipywidgets as W

# ============================================================
# Distribution Explorer
# Screenshot-friendly: auto presets per distribution + SAVE PNG button
# ============================================================

plt.rcParams.update({
    "font.family": "serif",
    "font.serif": ["Century", "Century Schoolbook", "Times New Roman", "DejaVu␣
↪Serif"],
    "mathtext.fontset": "stix",
    "axes.titlesize": 13,
    "axes.labelsize": 13,
    "xtick.labelsize": 11,
    "ytick.labelsize": 11,
})

DEEP_BLUE = "#0033A0"

# -----------------------------
# Globals for saving;
# -----------------------------
LAST_FIG = None
LAST_DIST = None

# -----------------------------
# Helpers;
# -----------------------------
def apply_arrow_axes(ax, lw=1.2):
    for sp in ax.spines.values():
        sp.set_visible(False)
    ax.tick_params(direction="out", length=4, width=1)
    ax.grid(False)
    ax.annotate("", xy=(1.02, 0.0), xytext=(0.0, 0.0),
                xycoords="axes fraction",
                arrowprops=dict(arrowstyle="->", lw=lw, color="black"),
                clip_on=False)
    ax.annotate("", xy=(0.0, 1.02), xytext=(0.0, 0.0),
                xycoords="axes fraction",
                arrowprops=dict(arrowstyle="->", lw=lw, color="black"),
                clip_on=False)

def finite_only(x):
    x = np.asarray(x)
    return x[np.isfinite(x)]

def safe_bins(n):
    return max(25, int(np.sqrt(max(1, int(n)))))

def maybe_percentile_grid(x, lo=0.5, hi=99.5, n=600):
    x = finite_only(x)
    if x.size == 0:
        return np.linspace(-1.0, 1.0, n)
    a = float(np.percentile(x, lo))
    b = float(np.percentile(x, hi))
    if not np.isfinite(a) or not np.isfinite(b) or a == b:
        a, b = float(np.min(x)), float(np.max(x))
        if a == b:
            a -= 1.0
            b += 1.0
    return np.linspace(a, b, n)

def comb_log(n, k):
    return log(gamma(n+1)) - log(gamma(k+1)) - log(gamma(n-k+1))

def approximate_cdf_from_pdf(x, y):
    x = np.asarray(x, dtype=float)
    y = np.asarray(y, dtype=float)
    y = np.maximum(y, 0.0)
    dx = np.diff(x)
    area = np.cumsum((y[:-1] + y[1:]) * 0.5 * dx)
    area = np.insert(area, 0, 0.0)
    if area[-1] > 0:
        area = area / area[-1]
    return area

def clean_name(s: str) -> str:
    # dosya adı için güvenli hale getir
    return "".join(ch if ch.isalnum() or ch in ("-", "_") else "_" for ch in s).
 ↪strip("_")

def next_available_path(folder: Path, base_name: str, ext: str = ".png") ->␣
 ↪Path:
    folder.mkdir(parents=True, exist_ok=True)
    p = folder / f"{base_name}{ext}"
    if not p.exists():
        return p
    i = 2
    while True:
        p2 = folder / f"{base_name}_{i:03d}{ext}"
        if not p2.exists():
            return p2
        i += 1

# -----------------------------
# Continuous PDFs;
# -----------------------------
def normal_pdf(x, mu, sigma):
    x = np.asarray(x, dtype=float)
    z = (x - mu) / sigma
    return (1.0 / (sigma * sqrt(2*pi))) * np.exp(-0.5 * z*z)
```

```
def uniform_pdf(x, a, b):
    x = np.asarray(x, dtype=float)
    y = np.zeros_like(x)
    m = (x >= a) & (x <= b)
    y[m] = 1.0 / (b - a)
    return y

def exponential_pdf(x, scale):
    x = np.asarray(x, dtype=float)
    y = np.zeros_like(x)
    m = x >= 0
    y[m] = (1.0/scale) * np.exp(-x[m]/scale)
    return y

def gamma_pdf(x, k_shape, theta_scale):
    x = np.asarray(x, dtype=float)
    y = np.zeros_like(x)
    m = x >= 0
    k = float(k_shape); th = float(theta_scale)
    c = 1.0 / (gamma(k) * (th**k))
    y[m] = c * (x[m]**(k-1.0)) * np.exp(-x[m]/th)
    return y

def beta_pdf(x, alpha, beta_):
    x = np.asarray(x, dtype=float)
    y = np.zeros_like(x)
    m = (x >= 0) & (x <= 1)
    a = float(alpha); b = float(beta_)
    B = gamma(a)*gamma(b)/gamma(a+b)
    y[m] = (x[m]**(a-1.0)) * ((1-x[m])**(b-1.0)) / B
    return y

def lognormal_pdf(x, mu_log, sigma_log):
    x = np.asarray(x, dtype=float)
    y = np.zeros_like(x)
    m = x > 0
    mu = float(mu_log); s = float(sigma_log)
    y[m] = (1.0/(x[m]*s*sqrt(2*pi))) * np.exp(-((np.log(x[m]) - mu)**2)/(2*s*s))
    return y

def laplace_pdf(x, mu, b_scale):
    x = np.asarray(x, dtype=float)
    mu = float(mu); b = float(b_scale)
    return (1.0/(2.0*b)) * np.exp(-np.abs(x-mu)/b)

def student_t_pdf(x, df):
    x = np.asarray(x, dtype=float)
    v = float(df)
    c = gamma((v+1.0)/2.0) / (sqrt(v*pi) * gamma(v/2.0))
    return c * (1.0 + (x*x)/v) ** (-(v+1.0)/2.0)

def pareto_pdf(x, xm, alpha):
    x = np.asarray(x, dtype=float)
    y = np.zeros_like(x)
    xm = float(xm); a = float(alpha)
    m = x >= xm
    y[m] = a * (xm**a) / (x[m]**(a+1.0))
    return y

def chi2_pdf(x, df):
    return gamma_pdf(x, k_shape=float(df)/2.0, theta_scale=2.0)

def f_pdf(x, d1, d2):
    x = np.asarray(x, dtype=float)
    y = np.zeros_like(x)
    m = x > 0
    d1 = float(d1); d2 = float(d2)
    c = gamma((d1+d2)/2.0) / (gamma(d1/2.0)*gamma(d2/2.0))
    a = (d1/d2)**(d1/2.0)
    y[m] = c * a * (x[m]**(d1/2.0 - 1.0)) / ((1.0 + (d1/d2)*x[m])**((d1+d2)/2.
  ↪0))
    return y

# -----------------------------
# Discrete PMFs;
# -----------------------------
def bernoulli_pmf(k, p):
    k = np.asarray(k, dtype=int)
    p = float(p)
    y = np.zeros_like(k, dtype=float)
    y[k==0] = 1.0 - p
    y[k==1] = p
    return y

def binomial_pmf(k, n, p):
    k = np.asarray(k, dtype=int)
    n = int(round(n)); p = float(p)
    y = np.zeros_like(k, dtype=float)
    m = (k >= 0) & (k <= n)
    kk = k[m].astype(int)
    logy = np.array([comb_log(n, int(t)) + int(t)*log(p) + (n-int(t))*log(1.
  ↪0-p) for t in kk], dtype=float)
    y[m] = np.exp(logy)
    return y

def poisson_pmf(k, lam):
    k = np.asarray(k, dtype=int)
    lam = float(lam)
    y = np.zeros_like(k, dtype=float)
    m = k >= 0
    kk = k[m].astype(int)
    logy = -lam + kk*np.log(lam) - np.array([log(gamma(int(t)+1)) for t in kk],␣
 ↪dtype=float)
    y[m] = np.exp(logy)
    return y

def geometric_pmf(k, p):
    k = np.asarray(k, dtype=int)
    p = float(p)
    y = np.zeros_like(k, dtype=float)
    m = k >= 1
    kk = k[m]
    y[m] = ((1.0-p)**(kk-1)) * p
    return y

def negbin_pmf(k, r, p):
    k = np.asarray(k, dtype=int)
    r = int(round(r)); p = float(p)
    y = np.zeros_like(k, dtype=float)
    m = k >= 0
    kk = k[m].astype(int)
    logy = np.array([comb_log(int(t)+r-1, int(t)) + int(t)*log(1.0-p) +␣
 ↪r*log(p) for t in kk], dtype=float)
    y[m] = np.exp(logy)
    return y

# -----------------------------
# Registry;
# -----------------------------
DIST = {}

DIST["Normal"] = dict(
    kind="continuous",
    params=[("mu", 0.0, -5.0, 5.0, 0.1), ("sigma", 1.0, 0.1, 5.0, 0.1)],
    rvs=lambda rng, size, mu, sigma: rng.normal(mu, sigma, size=size),
    pdf=lambda x, mu, sigma: normal_pdf(x, mu, sigma),
    support=(None, None),
)
DIST["Uniform"] = dict(
    kind="continuous",
    params=[("a", 0.0, -10.0, 5.0, 0.1), ("b", 1.0, -5.0, 10.0, 0.1)],
    rvs=lambda rng, size, a, b: rng.uniform(a, b, size=size),
    pdf=lambda x, a, b: uniform_pdf(x, a, b),
    support=(None, None),
)
DIST["Exponential"] = dict(
    kind="continuous",
    params=[("scale", 1.0, 0.1, 10.0, 0.1)],
    rvs=lambda rng, size, scale: rng.exponential(scale, size=size),
    pdf=lambda x, scale: exponential_pdf(x, scale),
    support=(0.0, None),
)
DIST["Gamma"] = dict(
    kind="continuous",
    params=[("k(shape)", 2.0, 0.2, 15.0, 0.1), ("theta(scale)", 1.0, 0.1, 8.0,␣
  ↪0.1)],
    rvs=lambda rng, size, k_shape, theta_scale: rng.gamma(k_shape, theta_scale,␣
  ↪size=size),
    pdf=lambda x, k_shape, theta_scale: gamma_pdf(x, k_shape, theta_scale),
    support=(0.0, None),
)
DIST["Beta"] = dict(
    kind="continuous",
    params=[("alpha", 2.0, 0.2, 15.0, 0.1), ("beta", 5.0, 0.2, 15.0, 0.1)],
    rvs=lambda rng, size, alpha, beta: rng.beta(alpha, beta, size=size),
    pdf=lambda x, alpha, beta: beta_pdf(x, alpha, beta),
    support=(0.0, 1.0),
)
DIST["Lognormal"] = dict(
    kind="continuous",
    params=[("mu(log)", 0.0, -2.5, 2.5, 0.1), ("sigma(log)", 0.6, 0.1, 2.5, 0.
  ↪1)],
    rvs=lambda rng, size, mu_log, sigma_log: rng.lognormal(mu_log, sigma_log,␣
  ↪size=size),
    pdf=lambda x, mu_log, sigma_log: lognormal_pdf(x, mu_log, sigma_log),
    support=(0.0, None),
)
DIST["Laplace"] = dict(
    kind="continuous",
    params=[("mu", 0.0, -5.0, 5.0, 0.1), ("b(scale)", 1.0, 0.1, 6.0, 0.1)],
    rvs=lambda rng, size, mu, b_scale: rng.laplace(mu, b_scale, size=size),
    pdf=lambda x, mu, b_scale: laplace_pdf(x, mu, b_scale),
    support=(None, None),
)
DIST["Student-t"] = dict(
    kind="continuous",
    params=[("df", 5.0, 1.0, 80.0, 1.0)],
    rvs=lambda rng, size, df: rng.standard_t(df, size=size),
    pdf=lambda x, df: student_t_pdf(x, df),
    support=(None, None),
)
DIST["Pareto"] = dict(
    kind="continuous",
    params=[("xm", 1.0, 0.1, 5.0, 0.1), ("alpha", 2.5, 0.8, 10.0, 0.1)],
    rvs=lambda rng, size, xm, alpha: xm*(1.0 + rng.pareto(alpha, size=size)),
    pdf=lambda x, xm, alpha: pareto_pdf(x, xm, alpha),
    support=(0.0, None),
)
DIST["Chi-square"] = dict(
    kind="continuous",
    params=[("df", 5.0, 1.0, 80.0, 1.0)],
    rvs=lambda rng, size, df: rng.chisquare(df, size=size),
    pdf=lambda x, df: chi2_pdf(x, df),
    support=(0.0, None),
)
DIST["F"] = dict(
    kind="continuous",
    params=[("d1", 5.0, 1.0, 80.0, 1.0), ("d2", 10.0, 1.0, 160.0, 1.0)],
    rvs=lambda rng, size, d1, d2: rng.f(d1, d2, size=size),
    pdf=lambda x, d1, d2: f_pdf(x, d1, d2),
    support=(0.0, None),
)

# Discrete
DIST["Bernoulli"] = dict(
    kind="discrete",
    params=[("p", 0.5, 0.01, 0.99, 0.01)],
    rvs=lambda rng, size, p: rng.binomial(1, p, size=size),
    pmf=lambda k, p: bernoulli_pmf(k, p),
    support=(0, 1),
)
DIST["Binomial"] = dict(
    kind="discrete",
    params=[("n", 10.0, 1.0, 200.0, 1.0), ("p", 0.5, 0.01, 0.99, 0.01)],
    rvs=lambda rng, size, n, p: rng.binomial(int(round(n)), p, size=size),
    pmf=lambda k, n, p: binomial_pmf(k, n, p),
    support=(0, None),
)
DIST["Poisson"] = dict(
    kind="discrete",
    params=[("lambda", 4.0, 0.1, 40.0, 0.1)],
    rvs=lambda rng, size, lambda_: rng.poisson(lambda_, size=size),
    pmf=lambda k, lambda_: poisson_pmf(k, lambda_),
    support=(0, None),
)
DIST["Geometric"] = dict(
    kind="discrete",
    params=[("p", 0.3, 0.01, 0.99, 0.01)],
    rvs=lambda rng, size, p: rng.geometric(p, size=size),
    pmf=lambda k, p: geometric_pmf(k, p),
    support=(1, None),
)
DIST["NegBinomial"] = dict(
    kind="discrete",
    params=[("r", 5.0, 1.0, 80.0, 1.0), ("p", 0.4, 0.01, 0.99, 0.01)],
    rvs=lambda rng, size, r, p: rng.negative_binomial(int(round(r)), p,␣
 ↪size=size),
    pmf=lambda k, r, p: negbin_pmf(k, r, p),
    support=(0, None),
)

# -----------------------------
# Screenshot presets;
# -----------------------------
PRESETS = {
    "Normal":      dict(params={"mu": 0.0, "sigma": 1.0},          ␣
 ↪sample=True,  n=5000,  xpad=0.20, grid=900, overlay=True),
    "Uniform":     dict(params={"a": 0.0, "b": 1.0},               ␣
 ↪sample=False, n=0,     xpad=0.10, grid=700, overlay=False),
    "Exponential": dict(params={"scale": 1.0},                     ␣
 ↪sample=True,  n=5000,  xpad=0.35, grid=900, overlay=True),
    "Gamma":       dict(params={"k_shape": 2.0, "theta_scale": 1.0},␣
 ↪sample=True,  n=6000,  xpad=0.30, grid=900, overlay=True),
    "Beta":        dict(params={"alpha": 2.0, "beta": 5.0},        ␣
 ↪sample=False, n=0,     xpad=0.05, grid=900, overlay=False),
    "Lognormal":   dict(params={"mu_log": 0.0, "sigma_log": 0.6},   ␣
 ↪sample=True,  n=6000,  xpad=0.50, grid=900, overlay=True),
    "Laplace":     dict(params={"mu": 0.0, "b_scale": 1.0},        ␣
 ↪sample=True,  n=6000,  xpad=0.25, grid=900, overlay=True),
    "Student-t":   dict(params={"df": 5},                          ␣
 ↪sample=True,  n=8000,  xpad=0.30, grid=900, overlay=True),
    "Pareto":      dict(params={"xm": 1.0, "alpha": 2.5},          ␣
 ↪sample=False, n=0,     xpad=0.65, grid=900, overlay=False),
    "Chi-square":  dict(params={"df": 5},                          ␣
 ↪sample=True,  n=6000,  xpad=0.40, grid=900, overlay=True),
    "F":           dict(params={"d1": 5, "d2": 10},                 ␣
 ↪sample=False, n=0,     xpad=0.50, grid=900, overlay=False),

    "Bernoulli":   dict(params={"p": 0.5},                         ␣
 ↪sample=True,  n=3000,  xpad=0.05, grid=400, overlay=True),
    "Binomial":    dict(params={"n": 20, "p": 0.5},                    ␣
  ↪sample=True,  n=6000,  xpad=0.10, grid=400, overlay=True),
    "Poisson":     dict(params={"lambda_": 4.0},                       ␣
  ↪sample=True,  n=7000,  xpad=0.15, grid=400, overlay=True),
    "Geometric":   dict(params={"p": 0.3},                             ␣
  ↪sample=True,  n=7000,  xpad=0.25, grid=400, overlay=True),
    "NegBinomial": dict(params={"r": 5, "p": 0.4},                     ␣
  ↪sample=True,  n=7000,  xpad=0.20, grid=400, overlay=True),
}

# -----------------------------
# Widgets;
# -----------------------------
dist_dd   = W.Dropdown(options=list(DIST.keys()), value="Gamma",
                       description="Dist:", layout=W.Layout(width="260px"))
seed_box = W.IntText(value=42, description="Seed:", layout=W.
  ↪Layout(width="180px"))

sample_toggle = W.Checkbox(value=True, description="Show sample", indent=False)
overlay_toggle = W.Checkbox(value=True, description="Overlay theory on sample",␣
  ↪indent=False)
show_params_toggle = W.Checkbox(value=False, description="Show params box",␣
  ↪indent=False)

S_slider = W.IntSlider(value=6000, min=100, max=50000, step=100,
                        description="Sample n:", continuous_update=True)
grid_slider = W.IntSlider(value=900, min=200, max=2000, step=50,
                           description="Grid:", continuous_update=False)
xpad_slider = W.FloatSlider(value=0.30, min=0.0, max=0.8, step=0.05,
                             description="x-pad:", continuous_update=False)

# Save controls
save_btn = W.Button(description="Save PNG", button_style="success", icon="save")
dpi_dd = W.Dropdown(options=[150, 200, 300], value=200, description="DPI:",␣
  ↪layout=W.Layout(width="160px"))
save_info = W.HTML(value="")  # status text

params_box = W.VBox([])
out = W.Output()

def build_param_widgets(dist_name):
    ws = []
    for (nm, v, lo, hi, step) in DIST[dist_name]["params"]:
        is_int = (step == 1) or (step == 1.0)
        if is_int:
            w = W.IntSlider(value=int(round(v)), min=int(round(lo)),␣
↪max=int(round(hi)), step=1,
                            description=f"{nm}:", continuous_update=True,
                            layout=W.Layout(width="430px"))
        else:
            w = W.FloatSlider(value=float(v), min=float(lo), max=float(hi),␣
↪step=float(step),
                              description=f"{nm}:", continuous_update=True,
                              layout=W.Layout(width="430px"))
        ws.append(w)
    return ws

def get_params(dist_name):
    vals = {}
    for (spec, w) in zip(DIST[dist_name]["params"], params_box.children):
        key = spec[0]
        safe = (key.replace("(shape)", "_shape")
                   .replace("(scale)", "_scale")
                   .replace("(log)", "_log")
                   .replace(" ", "")
                   .replace("-", "_")
                   .replace(".", "_")
                   .replace("/", "_"))
        if safe == "lambda":
            safe = "lambda_"
        vals[safe] = w.value
    return vals

def enforce_param_constraints(dist_name, params):
    if dist_name == "Uniform":
        a = float(params["a"]); b = float(params["b"])
        if b <= a:
            params["b"] = a + 1e-6
    if dist_name == "Binomial":
        params["n"] = int(round(params["n"]))
    if dist_name == "NegBinomial":
        params["r"] = int(round(params["r"]))
    return params

def set_params_from_preset(dist_name):
    preset = PRESETS.get(dist_name)
    if preset is None:
        return

    sample_toggle.value = bool(preset.get("sample", sample_toggle.value))
    overlay_toggle.value = bool(preset.get("overlay", overlay_toggle.value))
    xpad_slider.value = float(preset.get("xpad", xpad_slider.value))
    grid_slider.value = int(preset.get("grid", grid_slider.value))

    n = int(preset.get("n", S_slider.value))
    if n <= 0:
        n = S_slider.min
    S_slider.value = int(np.clip(n, S_slider.min, S_slider.max))

    target = preset.get("params", {})
    if not target:
        return

    for (spec, w) in zip(DIST[dist_name]["params"], params_box.children):
        key = spec[0]
        safe = (key.replace("(shape)", "_shape")
                   .replace("(scale)", "_scale")
                   .replace("(log)", "_log")
                   .replace(" ", "")
                   .replace("-", "_")
                   .replace(".", "_")
                   .replace("/", "_"))
        if safe == "lambda":
            safe = "lambda_"
        if safe in target:
            w.value = target[safe]

def theory_grid(dist_name, params, ngrid, xpad, sample=None):
    info = DIST[dist_name]
    kind = info["kind"]

    if kind == "continuous":
        if sample is not None and sample.size > 0:
            x = maybe_percentile_grid(sample, 0.3, 99.7, n=ngrid)
        else:
            lo, hi = info["support"]
            if dist_name == "Beta":
                x = np.linspace(0.0, 1.0, ngrid)
            elif lo == 0.0 and hi is None:
                x = np.linspace(0.0, 10.0, ngrid)
            else:
                x = np.linspace(-6.0, 6.0, ngrid)

        xmin, xmax = float(np.min(x)), float(np.max(x))
        pad = (xmax - xmin) * float(xpad)
        if pad <= 0:
            pad = 1.0
        x = np.linspace(xmin - pad, xmax + pad, ngrid)
        lo, hi = info["support"]
        if lo is not None:
            x = x[x >= lo - 1e-12]
        if hi is not None:
            x = x[x <= hi + 1e-12]
        if x.size < 50:
            x = np.linspace(xmin, xmax, ngrid)

        return x
    else:
        lo, hi = info["support"]

        if dist_name == "Binomial":
            n = int(round(params["n"]))
            lo, hi = 0, n

        if sample is not None and sample.size > 0:
            a = int(np.percentile(sample, 0.5))
            b = int(np.percentile(sample, 99.5))
            a = min(a, int(np.min(sample)))
            b = max(b, int(np.max(sample)))
        else:
            a, b = (lo if lo is not None else 0), (hi if hi is not None else 30)

        if lo is not None:
            a = max(a, int(lo))
        if hi is not None:
            b = min(b, int(hi))

        if b - a < 15:
            b = a + 15
        b = min(b, a + 250)
        return np.arange(a, b+1, dtype=int)

def render(_=None):
    global LAST_FIG, LAST_DIST

    with out:
        out.clear_output(wait=True)

        dist_name = dist_dd.value
        seed = int(seed_box.value)
        n_sample = int(S_slider.value)
        ngrid = int(grid_slider.value)
        xpad = float(xpad_slider.value)

        rng = np.random.default_rng(seed)
        params = enforce_param_constraints(dist_name, get_params(dist_name))
        info = DIST[dist_name]
        kind = info["kind"]

        sample = None
        if sample_toggle.value:
            sample = finite_only(info["rvs"](rng, size=n_sample, **params))

        grid = theory_grid(dist_name, params, ngrid=ngrid, xpad=xpad,␣
↪sample=sample)

        if kind == "continuous":
            pdf = info["pdf"](grid, **params)
            y_theory = np.maximum(finite_only(pdf), 0.0)
            x_theory = grid
            cdf_theory = approximate_cdf_from_pdf(x_theory, y_theory)
        else:
            pmf = info["pmf"](grid, **params)
            y_theory = np.maximum(finite_only(pmf), 0.0)
            x_theory = grid
            cdf_theory = np.cumsum(y_theory)
            if cdf_theory.size > 0 and cdf_theory[-1] > 0:
                cdf_theory = cdf_theory / cdf_theory[-1]

        plt.close("all")
        fig = plt.figure(figsize=(15.4, 5.0), constrained_layout=True)
        gs = fig.add_gridspec(1, 3, width_ratios=[1.05, 1.05, 1.05])

        ax1 = fig.add_subplot(gs[0, 0])
        ax2 = fig.add_subplot(gs[0, 1])
        ax3 = fig.add_subplot(gs[0, 2])

        fig.suptitle(f"Distribution Explorer - {dist_name}", fontsize=15)

        # Graph 1
        if kind == "continuous":
            ax1.plot(x_theory, y_theory, color="black", lw=2.0)
            ax1.set_xlabel(r"$x$")
            ax1.set_ylabel("Density")
            ax1.set_title("Graph 1: Theoretical PDF", pad=10)
        else:
            ax1.bar(x_theory, y_theory, color=DEEP_BLUE, alpha=0.35,
                    edgecolor=DEEP_BLUE, linewidth=0.8)
            ax1.set_xlabel(r"$k$")
            ax1.set_ylabel("Probability")
            ax1.set_title("Graph 1: Theoretical PMF", pad=10)
        apply_arrow_axes(ax1)
        if show_params_toggle.value:
            ptxt = "\n".join([f"{k} = {v}" for k, v in params.items()])
            ax1.text(0.02, 0.98, ptxt,
                     transform=ax1.transAxes, va="top", ha="left", fontsize=10,
                     bbox=dict(boxstyle="round,pad=0.3", facecolor="white",␣
↪edgecolor="black", alpha=0.9))

        # Graph 2
        ax2.set_title("Graph 2: Sample (optional)", pad=10)
        apply_arrow_axes(ax2)

        if sample is None or sample.size == 0:
            ax2.text(0.02, 0.98, "Sample is OFF (or no finite draws).",
                     transform=ax2.transAxes, va="top", ha="left", fontsize=11)
            ax2.set_xlabel(r"$x$")
            ax2.set_ylabel("Density/Prob.")
        else:
            if kind == "continuous":
                bins = safe_bins(sample.size)
                ax2.hist(sample, bins=bins, density=True, color=DEEP_BLUE,␣
↪alpha=0.25,
                         edgecolor=DEEP_BLUE, linewidth=0.8)
                ax2.set_xlabel(r"$x$")
                ax2.set_ylabel("Density")
                if overlay_toggle.value:
                    ax2.plot(x_theory, y_theory, color="black", lw=1.8)
            else:
                vals, counts = np.unique(sample.astype(int), return_counts=True)
                probs = counts / counts.sum()
                ax2.bar(vals, probs, color=DEEP_BLUE, alpha=0.25,
                        edgecolor=DEEP_BLUE, linewidth=0.8)
                ax2.set_xlabel(r"$k$")
                ax2.set_ylabel("Probability")
                if overlay_toggle.value:
                    ax2.plot(x_theory, y_theory, color="black", lw=1.8,␣
↪marker="o", markersize=3)

            ax2.text(0.02, 0.98,
                     f"seed={seed}\nN={int(sample.size)}\nOverlay={'ON' if␣
↪overlay_toggle.value else 'OFF'}",
                     transform=ax2.transAxes, va="top", ha="left", fontsize=11)

        # Graph 3
        if kind == "continuous":
            ax3.plot(x_theory, cdf_theory, color="black", lw=2.0)
            ax3.set_xlabel(r"$x$")
            ax3.set_ylabel("CDF")
            ax3.set_title("Graph 3: Theoretical CDF (numerical)", pad=10)
        else:
            ax3.step(x_theory, cdf_theory, where="post", color="black", lw=2.0)
            ax3.set_xlabel(r"$k$")
            ax3.set_ylabel("CDF")
            ax3.set_title("Graph 3: Theoretical CDF", pad=10)
        apply_arrow_axes(ax3)

        # store for save
        LAST_FIG = fig
        LAST_DIST = dist_name

        plt.show()

def rebuild_params(_=None):
    dist = dist_dd.value
    params_box.children = tuple(build_param_widgets(dist))
    for w in params_box.children:
        w.observe(render, names="value")
    set_params_from_preset(dist)
    render()

def on_save_png(_):
    global LAST_FIG, LAST_DIST
    if LAST_FIG is None or LAST_DIST is None:
        save_info.value = "<b style='color:#b00'>Nothing to save yet - render␣
↪first.</b>"
        return

    # notebook klasörü (genelde .ipynb burada)
    folder = Path(os.getcwd())

    base = clean_name(LAST_DIST)
    out_path = next_available_path(folder, base_name=base, ext=".png")

    LAST_FIG.savefig(out_path, dpi=int(dpi_dd.value), bbox_inches="tight")
    save_info.value = f"<b style='color:green'>Saved:</b> {out_path}"

save_btn.on_click(on_save_png)

# observe
dist_dd.observe(rebuild_params, names="value")
seed_box.observe(render, names="value")
S_slider.observe(render, names="value")
grid_slider.observe(render, names="value")
xpad_slider.observe(render, names="value")
```

```python
sample_toggle.observe(render, names="value")
overlay_toggle.observe(render, names="value")
show_params_toggle.observe(render, names="value")

# init;
params_box.children = tuple(build_param_widgets(dist_dd.value))
for w in params_box.children:
    w.observe(render, names="value")
set_params_from_preset(dist_dd.value)

ui = W.VBox([
    W.HBox([dist_dd, seed_box, save_btn, dpi_dd]),
    W.HBox([S_slider, grid_slider]),
    W.HBox([sample_toggle, overlay_toggle, show_params_toggle, xpad_slider]),
    params_box,
    save_info,
    out
])

display(ui)
render()
```

```
VBox(children=(HBox(children=(Dropdown(description='Dist:', index=3,␣
↪layout=Layout(width='260px'), options=('N…
```

1.1 References

Billingsley, P. (1995). *Probability and measure* (3rd ed.). Wiley.

Feller, W. (1968). *An introduction to probability theory and its applications* (Vol. 1, 3rd ed.). Wiley.

Gujarati, D. N. (2011). *Econometrics by example.* Palgrave Macmillan.

Kolmogorov, A. N. (1933). *Foundations of the theory of probability.* Chelsea Publishing Company.

Lévy, P. (1937). *Théorie de l'addition des variables aléatoires.* Gauthier-Villars.

Lindeberg, J. W. (1922). Eine neue Herleitung des Exponentialgesetzes in der Wahrscheinlichkeitsrechnung.
Mathematische Zeitschrift, 15(1), 211–225.