i need your help in creating heirarchical model .

Got it 👍 You want help in **creating a hierarchical model**. Before I dive in, let me clarify:

Hierarchical models (also called **multilevel models** or **random-effects models**) are useful when your data has **nested structure** — for example:

- Students nested in classes, classes in schools
- Properties nested in markets, markets in regions (like your real estate project)
- Patients nested in hospitals, repeated measurements nested within individuals

To guide you better, can you tell me:

1. **Context** – What is your dataset about? (e.g., real estate, forecasting, causal inference, etc.)
2. **Outcome** – What do you want to predict or estimate? (continuous target like price, or categorical like success/failure, or causal effect size)
3. **Software / Framework** – Do you prefer Python (PyMC, Stan via CmdStanPy, scikit-learn style), R (lme4, brms), or just conceptual guidance?

This way, I can either:

- Build a **general template** (like a PyMC hierarchical model for regression), or
- Tailor a **domain-specific hierarchical model** for your dataset.

👉 Do you want me to start with a **generic example** (say, hierarchical linear regression) and then adapt to your use case?

Let me give you a situation,
I have national level market which is called United_state followed by 10 market level followed by 5 submarket for each market .
Now you can image the tree strucure where you have United_state as root node , the it has 10 children market , and finally each market has 5 sub market as children .

Nice — thanks for the clear tree description. Below I'll give you a practical, ready-to-run set of hierarchical model designs (concept + code) for the tree:

United_state (root)
→ 10 markets (market_0 .. market_9)
→ each market has 5 submarkets (so 50 submarkets total; nested under markets)

I'll assume a **continuous outcome `y`** (e.g., price, sales). If your outcome is binary or counts, I'll add alternatives at the end. I'll also assume you might want both **varying intercepts** and optionally **varying slopes** for one or more predictors. If you want a plain intercept-only multilevel model, you can drop the slope parts.

1) Model outline (mathematical)

Indexing:

- `i` indexes observations (rows).
- `m[i]` ∈ {0..9} is the market index for observation `i`.
- `s[i]` ∈ {0..49} is the submarket index for observation `i`. Each submarket has a known parent market `market_of_sub[s]`.

Basic 3-level hierarchical model (varying intercepts):

$$
\begin{aligned}
y_i &\sim \mathcal{N}(\mu_i, \sigma_y) \\
\mu_i &= \alpha_{US} + \alpha^{(M)}_{m[i]} + \alpha^{(S)}_{s[i]} + X_i\beta \\
\alpha_{US} &\sim \mathcal{N}(0, \sigma_{US}) \text{ (could be fixed 0)} \\
\alpha^{(M)}_m &\sim \mathcal{N}(0, \sigma_M) \quad m = 0, \dots, 9 \\
\alpha^{(S)}_s &\sim \mathcal{N}(0, \sigma_S) \quad s = 0, \dots, 49 \\
\sigma_y, \sigma_M, \sigma_S &> 0 \text{ (hyperpriors)}
\end{aligned}
$$

If you want **varying slopes** for a predictor `x1` at market level:

$$
\mu_i = \alpha_{US} + \alpha^{(M)}_{m[i]} + \alpha^{(S)}_{s[i]} + \left(\beta_0 + \beta^{(M)}_{m[i]}\right) x_{1,i} + X_{-1,i}\beta_{-1}
$$

with $\beta^{(M)}_m \sim \mathcal{N}(0, \sigma_{\beta_M})$. You can also model intercept and slope correlation via a multivariate normal at market level.

---

2) Data preparation notes

- Create integer indices:
  - `market_idx` ∈ {0..9}
  - `submarket_idx` ∈ {0..49}
- Also create a `market_of_sub` array mapping each submarket index to its parent market index.
- Center / standardize continuous predictors — helps sampling and interpretability.

Example mapping in pandas:

```python
# pseudo
df['market_idx'] = df['market'].astype('category').cat.codes
df['submarket_idx'] = df['market'].astype(str) + '_' + df['submarket'].astype(str)
df['submarket_idx'] = df['submarket_idx'].astype('category').cat.codes
# build market_of_sub: for each submarket id, what's the market id
market_of_sub =
df[['submarket_idx','market_idx']].drop_duplicates().sort_values('submarket_idx')
['market_idx'].values
```

3) PyMC (Bayesian) implementation (full example)

This is a practical PyMC model implementing varying intercepts at market and submarket and an optional varying slope for `x1` at market level.

```python
# PyMC example (PyMC v4+/aesara)
import numpy as np
import pymc as pm
import arviz as az

# Inputs you must prepare:
# y: (N,) numpy array
# x1: (N,) predictor (centered)
# X_other: (N, P) other predictors (already standardized) or None
# market_idx: (N,) ints in 0..9
# submarket_idx: (N,) ints in 0..49
# market_of_sub: (S,) ints linking submarket->market (length S=50)

N = len(y)
n_markets = max(market_idx) + 1        # 10
n_submarkets = max(submarket_idx) + 1  # 50

with pm.Model() as model:
    # Hyperpriors for SDs
    sigma_y = pm.HalfNormal("sigma_y", sigma=1.0)
    sigma_M = pm.HalfNormal("sigma_M", sigma=1.0)
    sigma_S = pm.HalfNormal("sigma_S", sigma=1.0)

    # Global intercept (alpha_US)
    alpha_US = pm.Normal("alpha_US", mu=0.0, sigma=5.0)

    # Market-level varying intercepts
    alpha_M_raw = pm.Normal("alpha_M_raw", mu=0.0, sigma=1.0, shape=n_markets)
    alpha_M = pm.Deterministic("alpha_M", alpha_M_raw * sigma_M)

    # Submarket-level varying intercepts (centered per usual)
    alpha_S_raw = pm.Normal("alpha_S_raw", mu=0.0, sigma=1.0, shape=n_submarkets)
    alpha_S = pm.Deterministic("alpha_S", alpha_S_raw * sigma_S)

    # Fixed effect for x1
    beta_x1 = pm.Normal("beta_x1", mu=0.0, sigma=1.0)

    # Optional: varying slope for x1 at market level (uncomment to use)
    use_varying_slope = True
    if use_varying_slope:
        sigma_beta_M = pm.HalfNormal("sigma_beta_M", sigma=1.0)
        beta_M_raw = pm.Normal("beta_M_raw", 0.0, 1.0, shape=n_markets)
        beta_M = pm.Deterministic("beta_M", beta_M_raw * sigma_beta_M)
    else:
        beta_M = np.zeros(n_markets)

    # Optional additional fixed covariates
    # X_other: shape (N, P), beta_other shape (P,)
    # beta_other = pm.Normal('beta_other', mu=0, sigma=1, shape=P)

    # Expected value
    mu = (alpha_US
          + alpha_M[market_idx]
          + alpha_S[submarket_idx]
          + (beta_x1 + (beta_M[market_idx] if use_varying_slope else 0.0)) * x1
         )
    # add other fixed covariates: + X_other.dot(beta_other)

    y_obs = pm.Normal("y_obs", mu=mu, sigma=sigma_y, observed=y)

    # Sample
    trace = pm.sample(1000, tune=1000, target_accept=0.9)
    az.summary(trace, var_names=
["alpha_US","sigma_y","sigma_M","sigma_S","beta_x1","sigma_beta_M"])
```

Notes:

- **`alpha_M_raw`** and **`alpha_S_raw`** are standard-normal and scaled — common practice for hierarchical priors.
- If you want correlation between market intercept and slope, replace **`alpha_M_raw`** and **`beta_M_raw`** with a bivariate MVN with a 2×2 covariance (I can show that if you want).

---

4) lme4 (R) — quick formula if you prefer frequentist

If you prefer **`lme4`** in R, a simple random intercepts model:

```
r

# intercepts for market and submarket nested
lmer(y ~ x1 + (1 | market) + (1 | market:submarket), data = df)

# if varying slope for x1 at market-level:
lmer(y ~ x1 + (1 + x1 | market) + (1 | market:submarket), data = df)
```

`market:submarket` enforces nesting. You can also use **`(1 | submarket)`** if **`submarket`** IDs are globally unique and already encode parent market (i.e., submarket IDs aren't repeated across markets).

---

5) Stan (for multivariate market intercept/slope correlation)

If you want correlation between intercept and slope at market-level, Stan multivariate normal is typical. Sketch (pseudo):

```
cpp

data { int<lower=1> N; int<lower=1> M; int<lower=1> S;
       int market_idx[N]; int submarket_idx[N];
       vector[N] x1; vector[N] y; int market_of_sub[S]; }
parameters {
  real alpha_us;
  vector[2] mu_M; // not needed
  cholesky_factor_corr[2] L_Omega_M;
  vector<lower=0>[2] tau_M;
  matrix[2,M] z_M; // raw
  vector[S] alpha_S_raw;
  real<lower=0> sigma_S;
  real<lower=0> sigma_y;
  real beta_global;
}
transformed parameters {
  matrix[2,M] b_M = diag_pre_multiply(tau_M, L_Omega_M) * z_M;
  // b_M[1,m] intercept dev, b_M[2,m] slope dev
}
model {
  // priors ...
  for (n in 1:N)
    y[n] ~ normal(alpha_us + b_M[1, market_idx[n]] + alpha_S_raw[submarket_idx[n]]
                 + (beta_global + b_M[2, market_idx[n]]) * x1[n], sigma_y);
}
```

Tell me if you want a full Stan file and I'll paste a copy.

---

6) Practical tips & diagnostics

- **Center continuous predictors** (especially x1) — reduces correlation between intercept and slope.
- **Priors:** set sensible weakly-informative priors (e.g., normal(0,1) for standardized predictors). Use HalfNormal(1) for scale hyperpriors.
- **Check pooling:** inspect posterior SDs `sigma_M`, `sigma_S`. If `sigma_S` ≈ 0, submarket-level adds little.
- **Posterior predictive checks** (ppc) — always run `az.plot_ppc` / `pm.sample_posterior_predictive` to compare predicted vs observed.
- **Model comparison:** compare simpler (no varying slopes) vs richer models with LOO/WAIC or leave-one-market-out checks.
- **If many predictors**: consider group-level regularization (horseshoe, hierarchical shrinkage) or dimensionality reduction.
- **If time series**: extend to include temporal correlation (AR terms) at appropriate levels.

---

7) Binary outcome / counts

- Binary: replace `y_obs ~ Normal` with `pm.Bernoulli("y_obs", logit_p=logit_p, observed=y)` where `logit_p = mu` and `mu` becomes linear predictor.
- Counts: use `pm.Poisson` or `pm.NegativeBinomial` with `log(mu) = linear_predictor`.

---

8) Next steps I can do for you (pick any — no need to answer now if you already know)

- Produce a fully runnable PyMC notebook with simulated example data (I can include simulation code and sampling commands).
- Provide a full Stan `.stan` file ready to compile.
- Convert to a glmer/lme4-ready R script and show diagnostics there.
- Add correlated random effects (market intercept–slope covariance) example in PyMC or Stan.
- Adapt the model for binary/count outcome or include time component.

If you want, I can **immediately** produce a runnable PyMC notebook that simulates data for your exact tree (United_state → 10 markets → 5 submarkets each), fits the model, and shows diagnostic plots and summaries. Or I can produce the full Stan file. Which one would you like me to generate now?

i am looking for both varying intercepts and optionally varying slopes for one or more predictors.

Great — I’ve prepared two ready-to-run artifacts you can use **right away**:

- a **PyMC (Python) script** that (1) simulates data for your tree (United_state → 10 markets → 5 submarkets each), (2) fits a hierarchical model with **varying intercepts** at market & submarket, and **optionally varying slopes** for one or more predictors (with correlated intercept–slope at the market level), and (3) prints diagnostics and summaries; and
- a **Stan model** (.stan) implementing the same structure (market-level correlated intercept & slope, submarket intercepts).

I’m dropping both artifacts below so you can copy-paste and run them. If you want, I can also run the PyMC code here and show results, or tailor priors/predictors to your exact variables — tell me which predictors you care about and I’ll adapt.

PyMC hierarchical model for United_state -> 10 markets -> 5 submarkets each

- Simulates data

- Fits varying intercepts (market, submarket)

- Optionally fits varying slopes at market level (with intercept-slope correlation)

Requirements:

pip install pymc arviz numpy pandas

Usage:

python pymc_hierarchical_us_markets.py

```
import numpy as np
import pandas as pd
import pymc as pm
import arviz as az
```

```
RANDOM_SEED = 123
np.random.seed(RANDOM_SEED)
```

Simulation parameters

----

n_markets = 10
n_submarkets_per_market = 5
n_submarkets = n_markets * n_submarkets_per_market
n_per_submarket = 100 # observations per submarket
N = n_submarkets * n_per_submarket

True generative parameters (for simulated data)

true_alpha_US = 1.0

Market-level true intercept std and slope std (for sim)

true_sigma_M_intercept = 0.8
true_sigma_M_slope = 0.5
true_rho = 0.3 # correlation between intercept and slope at market level

Submarket-level intercept std

true_sigma_S = 0.4

Observation noise

true_sigma_y = 1.0

Build hierarchical indices

----

market_ids = np.repeat(np.arange(n_markets), n_submarkets_per_market * n_per_submarket)
submarket_local_idx = np.tile(np.repeat(np.arange(n_submarkets_per_market), n_per_submarket), n_markets)

Flatten submarket to global index 0..(n_submarkets-1)

submarket_ids = np.concatenate([
np.arange(m * n_submarkets_per_market, (m + 1) * n_submarkets_per_market)
for m in range(n_markets)
])
submarket_ids = np.repeat(submarket_ids, n_per_submarket)

Sanity: lengths should be N

```
assert len(market_ids) == N
assert len(submarket_ids) == N
```

Simulate predictors

Example predictor x1 (we'll allow varying slope across markets for x1)

```
x1 = np.random.normal(size=N)
```

Another predictor x2 (fixed slope)

```
x2 = np.random.normal(size=N)
```

Standardize predictors (helpful for sampling)

```
x1 = (x1 - x1.mean()) / x1.std()
x2 = (x2 - x2.mean()) / x2.std()
```

Simulate hierarchical parameters

Market-level bi-variate (intercept, slope) with correlation

```
cov_M = np.array([[true_sigma_M_intercept ** 2, true_rho * true_sigma_M_intercept * true_sigma_M_slope],
[true_rho * true_sigma_M_intercept * true_sigma_M_slope, true_sigma_M_slope ** 2]])
mu_M = np.array([0.0, 0.0])
market_b = np.random.multivariate_normal(mean=mu_M, cov=cov_M, size=n_markets)
market_intercept = market_b[:, 0]
market_slope_x1 = market_b[:, 1]
```

Submarket-level intercepts (centered around 0)

```
submarket_intercept = np.random.normal(loc=0.0, scale=true_sigma_S, size=n_submarkets)
```

Global fixed slope for x2

```
true_beta_x2 = 0.6
```

Build observation-level mean

```
mu = (
true_alpha_US
+ market_intercept[market_ids]
+ submarket_intercept[submarket_ids]
+ market_slope_x1[market_ids] * x1
+ true_beta_x2 * x2
)
```

```
y = mu + np.random.normal(scale=true_sigma_y, size=N)
```

Put into DataFrame (helpful)

```
df = pd.DataFrame({
"y": y,
"x1": x1,
"x2": x2,
"market_idx": market_ids,
"submarket_idx": submarket_ids
})
```

Build mapping market_of_sub: for each submarket index (0..S-1), which market is parent

```
market_of_sub = df[['submarket_idx', 'market_idx']].drop_duplicates().sort_values('submarket_idx')['market_idx'].values.astype(int)
```

PyMC model: varying intercepts + optional varying slopes

```
use_varying_slope = True # set False to disable market-level varying slope
```

```
coords = {
"obs_id": np.arange(N),
"market": np.arange(n_markets),
"submarket": np.arange(n_submarkets),
"coef": ["intercept", "slope_x1"] # for correlated b_M if used
}
```

```
with pm.Model(coords=coords) as hierarchical_model:
# Hyperpriors for noise
sigma_y = pm.HalfNormal("sigma_y", sigma=2.0)
```

```
# Global intercept
alpha_US = pm.Normal("alpha_US", mu=0.0, sigma=5.0)

# --------------------
# Market-level: correlated intercept & slope for x1
# --------------------
if use_varying_slope:
    # LKJ prior for correlation
    packed_L = pm.LKJCholeskyCov("packed_L", n=2, eta=2.0, sd_dist=pm.HalfNormal.dist(1.0))
    L = pm.expand_packed_triangular(2, packed_L)
    # raw market effects (2 x n_markets)
    z_market = pm.Normal("z_market", mu=0.0, sigma=1.0, shape=(2, n_markets))
    # actual b_M: (2 x n_markets)
    b_market = pm.Deterministic("b_market", (L @ z_market))
    # intercept and slope arrays
    alpha_M = pm.Deterministic("alpha_M", b_market[0, :])
    beta_M_x1 = pm.Deterministic("beta_M_x1", b_market[1, :])
else:
    # Only varying intercepts at market level
    sigma_alpha_M = pm.HalfNormal("sigma_alpha_M", sigma=1.0)
    alpha_M_raw = pm.Normal("alpha_M_raw", mu=0.0, sigma=1.0, shape=n_markets)
    alpha_M = pm.Deterministic("alpha_M", alpha_M_raw * sigma_alpha_M)
    # zero slope deviations
    beta_M_x1 = pm.Deterministic("beta_M_x1", np.zeros(n_markets))

# --------------------
# Submarket-level varying intercept (no varying slopes at submarket)
# --------------------
sigma_S = pm.HalfNormal("sigma_S", sigma=1.0)
alpha_S_raw = pm.Normal("alpha_S_raw", mu=0.0, sigma=1.0, shape=n_submarkets)
alpha_S = pm.Deterministic("alpha_S", alpha_S_raw * sigma_S)

# Fixed effect for x2
beta_x2 = pm.Normal("beta_x2", mu=0.0, sigma=1.0)

# Expected value
mu_pred = (
    alpha_US
    + alpha_M[df["market_idx"].values]
    + alpha_S[df["submarket_idx"].values]
    + beta_M_x1[df["market_idx"].values] * df["x1"].values
    + beta_x2 * df["x2"].values
)

# Likelihood
y_obs = pm.Normal("y_obs", mu=mu_pred, sigma=sigma_y, observed=df["y"].values)

# Sample
idata = pm.sample(draws=1000, tune=1000, target_accept=0.9, return_inferencedata=True)
```

Summaries & diagnostics

```
print(az.summary(idata, var_names=["alpha_US", "sigma_y", "sigma_S", "beta_x2"], round_to=2))
if use_varying_slope:
print("\nMarket-level intercept/slope (posterior means):")
post_alpha_M = idata.posterior["alpha_M"].mean(dim=("chain", "draw")).values
post_beta_M_x1 = idata.posterior["beta_M_x1"].mean(dim=("chain", "draw")).values
```

```
print(pd.DataFrame({
"market": np.arange(n_markets),
"post_alpha_M": post_alpha_M,
"post_beta_M_x1": post_beta_M_x1
}))
else:
print("\nMarket-level intercepts (posterior means):")
post_alpha_M = idata.posterior["alpha_M"].mean(dim=("chain", "draw")).values
print(pd.DataFrame({
"market": np.arange(n_markets),
"post_alpha_M": post_alpha_M
}))
```

PPC

```
ppc = pm.sample_posterior_predictive(idata, model=hierarchical_model)
az.plot_ppc(az.from_pymc3(posterior_predictive=ppc, model=hierarchical_model)) # minor
compatibility helper
print("Done.")
```

```
// Stan model: 3-level hierarchical model
// - Market-level correlated intercept & slope for x1
// - Submarket-level varying intercepts (nested under market)
// Data needed (from R/PyStan/cmdstan):
// N: number of observations
// M: number of markets
// S: number of submarkets
// market_idx[N]: 1..M
// submarket_idx[N]: 1..S
// market_of_sub[S]: parent market for each submarket (1..M)
// x1[N], x2[N], y[N]
data {
int<lower=1> N;
int<lower=1> M;
int<lower=1> S;
int<lower=1,upper=M> market_idx[N];
int<lower=1,upper=S> submarket_idx[N];
int<lower=1,upper=M> market_of_sub[S];
vector[N] x1;
vector[N] x2;
vector[N] y;
}
parameters {
real alpha_US;
vector[2] mu_market; // prior mean for market effects (can be 0)
cholesky_factor_corr[2] L_Omega;
vector<lower=0>[2] tau_market; // scales for intercept & slope
matrix[2, M] z_market; // raw market effects
vector[S] alpha_S_raw; // raw submarket intercepts
real<lower=0> sigma_S; // scale for submarket intercepts
real beta_x2;
real<lower=0> sigma_y;
}
transformed parameters {
matrix[2,M] b_market;
for (m in 1:M) {
b_market[, m] = (diag_pre_multiply(tau_market, L_Omega) * z_market[, m]) + mu_market;
}
vector[M] alpha_M = b_market[1, ]'; // market intercept deviations
vector[M] beta_M_x1 = b_market[2, ]'; // market slope deviations for x1
vector[S] alpha_S = alpha_S_raw * sigma_S;
}
model {
// Priors
alpha_US ~ normal(0, 5);
mu_market ~ normal(0, 1);
L_Omega ~ lkj_corr_cholesky(2.0);
tau_market ~ cauchy(0, 2); // half-Cauchy-ish
to_vector(z_market) ~ normal(0, 1);
alpha_S_raw ~ normal(0, 1);
sigma_S ~ normal(0, 1);
beta_x2 ~ normal(0, 1);
sigma_y ~ normal(0, 1);

// Likelihood
for (n in 1:N) {
real mu_n = alpha_US
+ alpha_M[market_idx[n]]
+ alpha_S[submarket_idx[n]]
+ beta_M_x1[market_idx[n]] * x1[n]
+ beta_x2 * x2[n];
y[n] ~ normal(mu_n, sigma_y);
}
}
generated quantities {
corr_matrix[2] R_market;
R_market = multiply_lower_tri_self_transpose(L_Omega);
}
```

Can you please write the clean code in pymc ?

"""
Clean PyMC implementation: 3-level hierarchical model for
United_state -> Markets (10) -> Submarkets (5 per market).

Features:

- Varying intercepts at market and submarket levels
- Optional market-level varying slopes for one or more predictors
- Correlated intercept & slope at market level (when varying slopes enabled)
- Works with your own DataFrame or will simulate example data if none provided

Requirements:
pip install pymc arviz pandas numpy matplotlib

Usage:
- Import this file as a module and call `fit_hierarchical_model(df, ...)`
- Or run as script to simulate data and fit the model
"""

from typing import Optional, Sequence, Dict
import numpy as np
import pandas as pd
import pymc as pm
import arviz as az
import matplotlib.pyplot as plt

RNG_SEED = 123

def make_example_data(
n_markets: int = 10,
n_sub_per_market: int = 5,
n_per_submarket: int = 100,
seed: int = RNG_SEED,
) -> pd.DataFrame:
"""Simulate data for the hierarchical structure.

```
Returns DataFrame with columns:
  - y: continuous outcome
  - x1: predictor with market-level varying slope
  - x2: predictor with global fixed slope
  - market: categorical market label
  - submarket: categorical submarket label (unique across markets)
"""
rng = np.random.default_rng(seed)
n_submarkets = n_markets * n_sub_per_market
N = n_submarkets * n_per_submarket

# Build indices
market_idx = np.repeat(np.arange(n_markets), n_sub_per_market * n_per_submarket)
sub_idx_local = np.tile(np.repeat(np.arange(n_sub_per_market), n_per_submarket), n_markets)
# global submarket id 0..S-1
submarket_ids = np.concatenate(
    [np.arange(m * n_sub_per_market, (m + 1) * n_sub_per_market) for m in range(n_markets)]
)
submarket_idx = np.repeat(submarket_ids, n_per_submarket)

# Predictors
x1 = rng.normal(size=N)  # will have market-varying slope
```

```
x2 = rng.normal(size=N)  # fixed slope

# Standardize predictors
x1 = (x1 - x1.mean()) / x1.std()
x2 = (x2 - x2.mean()) / x2.std()

# True generative params (for sim)
alpha_US = 1.0
sigma_M_intercept = 0.8
sigma_M_slope = 0.5
rho = 0.3
sigma_S = 0.4
sigma_y = 1.0
beta_x2 = 0.6

# Market-level correlated intercept and slope
cov_M = np.array(
    [
        [sigma_M_intercept ** 2, rho * sigma_M_intercept * sigma_M_slope],
        [rho * sigma_M_intercept * sigma_M_slope, sigma_M_slope ** 2],
    ]
)
b_M = rng.multivariate_normal(mean=[0.0, 0.0], cov=cov_M, size=n_markets)
alpha_M_true = b_M[:, 0]
beta_M_x1_true = b_M[:, 1]

# Submarket intercepts
alpha_S_true = rng.normal(0, sigma_S, size=n_submarkets)

mu = (
    alpha_US
    + alpha_M_true[market_idx]
    + alpha_S_true[submarket_idx]
    + beta_M_x1_true[market_idx] * x1
    + beta_x2 * x2
)
y = mu + rng.normal(0, sigma_y, size=N)

df = pd.DataFrame(
    {
        "y": y,
        "x1": x1,
        "x2": x2,
        "market": pd.Categorical(market_idx).rename_categories(
            lambda i: f"market_{i}"
        ),
        "submarket": pd.Categorical(submarket_idx).rename_categories(
            lambda i: f"sub_{i}"
        ),
    }
)
# keep integer index columns for model
df["market_idx"] = df["market"].cat.codes.astype(int)
df["submarket_idx"] = df["submarket"].cat.codes.astype(int)
return df
```

```
def prepare_indices(df: pd.DataFrame, market_col: str = "market", sub_col: str = "submarket"):
"""Return integer indices and mapping (market_of_sub)."""
# Ensure categorical encoding
df = df.copy()
df[market_col] = df[market_col].astype("category")
df[sub_col] = df[sub_col].astype("category")
```

```
market_idx = df[market_col].cat.codes.values.astype(int)
submarket_idx = df[sub_col].cat.codes.values.astype(int)

n_markets = df[market_col].nunique()
n_submarkets = df[sub_col].nunique()

# mapping from submarket_id -> parent market id
mapping = (
    df[[sub_col, market_col]]
    .drop_duplicates()
    .sort_values(by=sub_col)
    .reset_index(drop=True)
)
market_of_sub = mapping[market_col].cat.codes.values.astype(int)

return {
    "market_idx": market_idx,
    "submarket_idx": submarket_idx,
    "n_markets": n_markets,
    "n_submarkets": n_submarkets,
    "market_of_sub": market_of_sub,
}
```

def fit_hierarchical_model(

df: pd.DataFrame,

outcome: str = "y",

varying_slope_predictors: Optional[Sequence[str]] = ("x1",),

fixed_predictors: Optional[Sequence[str]] = ("x2",),

use_varying_slope: bool = True,

draws: int = 1000,

tune: int = 1000,

random_seed: int = RNG_SEED,

) -> az.InferenceData:

"""Build and fit the hierarchical PyMC model.

```
- varying_slope_predictors: list of predictor names for which market-level varying slopes are
modeled.
  (correlated with market intercepts)
- fixed_predictors: list of predictor names with global fixed slopes.
- use_varying_slope: toggle the market-level varying slope block (if False, only varying
intercepts)
"""

# Prepare indices
idx = prepare_indices(df, market_col="market", sub_col="submarket")
market_idx = idx["market_idx"]
submarket_idx = idx["submarket_idx"]
n_markets = idx["n_markets"]
n_submarkets = idx["n_submarkets"]

# Outcome & predictors as numpy arrays
y = df[outcome].values
X_fixed = np.column_stack([df[c].values for c in (fixed_predictors or [])]) if fixed_predictors
else None

# For simplicity this implementation supports at most one varying-slope predictor.
if varying_slope_predictors is None:
    varying_slope_predictors = []
if len(varying_slope_predictors) > 1:
    raise NotImplementedError("This clean template implements at most one varying-slope
predictor (extendable).")

varying_pred = varying_slope_predictors[0] if len(varying_slope_predictors) == 1 else None
xv = df[varying_pred].values if varying_pred is not None else None

coords = {"market": np.arange(n_markets), "submarket": np.arange(n_submarkets), "obs_id":
np.arange(len(y))}

with pm.Model(coords=coords) as model:
    # Observation noise
    sigma_y = pm.HalfNormal("sigma_y", sigma=2.0)

    # Global intercept
    alpha_US = pm.Normal("alpha_US", mu=0.0, sigma=5.0)

    # ---------------------------
    # Market-level: intercept ± slope (correlated)
    # --------------------------
    if use_varying_slope and varying_pred is not None:
        # Two-dimensional (intercept, slope) per market with LKJ prior for correlation
        sd_dist = pm.HalfNormal.dist(1.0)
        packed = pm.LKJCholeskyCov("packed_L", n=2, eta=2.0, sd_dist=sd_dist)
        L = pm.expand_packed_triangular(2, packed)
        # raw normals
        z = pm.Normal("z_market", mu=0.0, sigma=1.0, shape=(2, n_markets))
        # correlated effects: (2, n_markets)
        b_market = pm.Deterministic("b_market", L @ z)
        alpha_M = pm.Deterministic("alpha_M", b_market[0, :])      # intercept deviations
        beta_M = pm.Deterministic("beta_M", b_market[1, :])        # slope deviations for
varying predictor
    else:
        # Only varying intercept at market
        sigma_alpha_M = pm.HalfNormal("sigma_alpha_M", sigma=1.0)
        alpha_M_raw = pm.Normal("alpha_M_raw", 0.0, 1.0, shape=n_markets)
        alpha_M = pm.Deterministic("alpha_M", alpha_M_raw * sigma_alpha_M)
        # create zero slope deviations
        beta_M = pm.Deterministic("beta_M", np.zeros(n_markets))

    # --------------------------
    # Submarket-level varying intercepts
    # --------------------------
    sigma_S = pm.HalfNormal("sigma_S", sigma=1.0)
    alpha_S_raw = pm.Normal("alpha_S_raw", 0.0, 1.0, shape=n_submarkets)
    alpha_S = pm.Deterministic("alpha_S", alpha_S_raw * sigma_S)

    # --------------------------
    # Fixed effects for fixed_predictors
    # --------------------------
    if X_fixed is not None:
        P = X_fixed.shape[1]
        beta_fixed = pm.Normal("beta_fixed", mu=0.0, sigma=1.0, shape=P)
    else:
        beta_fixed = None

    # --------------------------
    # Expected value
    # --------------------------
    mu = alpha_US + alpha_M[market_idx] + alpha_S[submarket_idx]

    if varying_pred is not None:
        mu = mu + beta_M[market_idx] * xv

    if X_fixed is not None:
        mu = mu + (X_fixed @ beta_fixed)

    # Likelihood
    y_obs = pm.Normal("y_obs", mu=mu, sigma=sigma_y, observed=y)

    # Sample
    idata = pm.sample(draws=draws, tune=tune, target_accept=0.9, random_seed=random_seed,
return_inferencedata=True)

return idata
```

```
def main():
# Example run: simulate data and fit
df = make_example_data()
print("Simulated data shape:", df.shape)
idata = fit_hierarchical_model(
df,
outcome="y",
varying_slope_predictors=("x1",), # market-level varying slope on x1
fixed_predictors=("x2",),
use_varying_slope=True,
```

draws=1000,
tune=1000,
)

```
print("\nSummary (selected):")
print(az.summary(idata, var_names=["alpha_US", "sigma_y", "sigma_S"], round_to=2))

# Posterior means for market intercepts & slopes (if present)
if "alpha_M" in idata.posterior:
    pm_alpha = idata.posterior["alpha_M"].mean(dim=("chain", "draw")).values
    print("\nPosterior mean - market intercept deviations (first 10):")
    print(pm_alpha[:10])

if "beta_M" in idata.posterior:
    pm_beta = idata.posterior["beta_M"].mean(dim=("chain", "draw")).values
    print("\nPosterior mean - market slope deviations for x1 (first 10):")
    print(pm_beta[:10])

# Simple PPC plot
az.plot_ppc(idata, var_names=["y_obs"], num_pp_samples=100)
plt.show()
```

if **name** == "**main**":
main()

Now you have one challenge , I want to use ML model instead of pymc since i am working with non - linear model .
I want to you implement this with ML approach which should replicate the exact things.

Nice — good challenge. Below I'll (1) explain the approach, then (2) give you **clean, ready-to-run PyTorch code** that implements an ML version of the hierarchical model you had in PyMC:

Idea / approach (short)

- We want to **replicate varying intercepts** and **varying slopes** within a flexible non-linear model.
- Use a neural network that learns:
  - **Market embeddings** and **Submarket embeddings** (these play the role of group-level random effects / learned offsets).
  - An explicit **market-level slope embedding** for predictor(s) where you want varying slopes. That embedding is used multiplicatively with that predictor so the network has a direct per-market slope term (replicates varying slopes).
  - The rest of predictors and embeddings are fed into an MLP so the model can learn non-linear interactions and complex effects.
- This keeps interpretability (we can inspect embedding vectors and the learned per-market slope) while allowing arbitrary non-linearities through the MLP.
- The code includes: data simulation matching your tree, PyTorch Dataset, model, training loop, extraction of per-market intercepts & slopes (posterior-like point estimates), and evaluation (MSE).

Below is the full code. Copy-paste and run in a Python environment with the listed packages.

""
ML-based hierarchical model in PyTorch that replicates:

- Varying intercepts at market and submarket level (via embeddings)
- Optionally varying slopes for one predictor (market-level slope embedding)
- Flexible non-linear modeling with an MLP

Requirements:
pip install torch numpy pandas scikit-learn matplotlib

Usage:
Run as script to simulate data, fit model, and inspect learned intercepts/slopes.
""

import math
import numpy as np
import pandas as pd
from typing import Optional, Tuple
import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import random

SEED = 1234
torch.manual_seed(SEED)
np.random.seed(SEED)
random.seed(SEED)

Data simulation (same tree)

----

def simulate_hierarchical_data(
n_markets: int = 10,
n_sub_per_market: int = 5,
n_per_submarket: int = 100,
include_varying_slope: bool = True,
noise_sigma: float = 1.0,
seed: int = SEED,
) -> pd.DataFrame:
rng = np.random.default_rng(seed)

```
n_submarkets = n_markets * n_sub_per_market
N = n_submarkets * n_per_submarket

# indices
market_idx = np.repeat(np.arange(n_markets), n_sub_per_market * n_per_submarket)
submarket_block = np.tile(np.repeat(np.arange(n_sub_per_market), n_per_submarket), n_markets)
```

```
submarket_global = np.concatenate(
    [np.arange(m * n_sub_per_market, (m + 1) * n_sub_per_market) for m in range(n_markets)]
)
submarket_idx = np.repeat(submarket_global, n_per_submarket)

# predictors
x1 = rng.normal(size=N)  # predictor with market-varying slope
x2 = rng.normal(size=N)  # predictor with global slope

# standardize predictors
x1 = (x1 - x1.mean()) / x1.std()
x2 = (x2 - x2.mean()) / x2.std()

# true generative params (for sim)
alpha_US = 1.0
sigma_M_intercept = 0.8
sigma_M_slope = 0.5
rho = 0.3
sigma_S = 0.4
beta_x2 = 0.6

# correlated market intercept & slope
cov_M = np.array(
    [
        [sigma_M_intercept ** 2, rho * sigma_M_intercept * sigma_M_slope],
        [rho * sigma_M_intercept * sigma_M_slope, sigma_M_slope ** 2],
    ]
)
b_M = rng.multivariate_normal(mean=[0.0, 0.0], cov=cov_M, size=n_markets)
alpha_M_true = b_M[:, 0]
beta_M_x1_true = b_M[:, 1] if include_varying_slope else np.zeros(n_markets)

alpha_S_true = rng.normal(0, sigma_S, size=n_submarkets)

mu = (
    alpha_US
    + alpha_M_true[market_idx]
    + alpha_S_true[submarket_idx]
    + beta_M_x1_true[market_idx] * x1
    + beta_x2 * x2
)

y = mu + rng.normal(0, noise_sigma, size=N)

df = pd.DataFrame(
    {
        "y": y,
        "x1": x1,
        "x2": x2,
        "market_idx": market_idx.astype(int),
        "submarket_idx": submarket_idx.astype(int),
    }
)
return df
```

PyTorch Dataset

----

class HierDataset(Dataset):

def **init**(self, df: pd.DataFrame, device: torch.device = torch.device("cpu")):

self.device = device

# convert to tensors

self.y = torch.tensor(df["y"].values, dtype=torch.float32, device=device).unsqueeze(1) self.x1 = torch.tensor(df["x1"].values, dtype=torch.float32, device=device).unsqueeze(1)

self.x2 = torch.tensor(df["x2"].values, dtype=torch.float32, device=device).unsqueeze(1)

self.market_idx = torch.tensor(df["market_idx"].values, dtype=torch.long, device=device)

self.submarket_idx = torch.tensor(df["submarket_idx"].values, dtype=torch.long, device=device)

```
def __len__(self):
    return len(self.y)

def __getitem__(self, idx):
    return {
        "y": self.y[idx],
        "x1": self.x1[idx],
        "x2": self.x2[idx],
        "market_idx": self.market_idx[idx],
        "submarket_idx": self.submarket_idx[idx],
    }
```

```
-----------------------
```

Model

```
-----------------------
```

class HierarchicalMLP(nn.Module):

def **init**(

self,

n_markets: int,

n_submarkets: int,

embed_dim_market: int = 8,

embed_dim_sub: int = 6,

mlp_hidden: Tuple[int, ...] = (64, 32),

use_market_varying_slope: bool = True,

):

super().**init**()

self.use_market_varying_slope = use_market_varying_slope

```
    # embeddings for intercepts
    self.market_emb = nn.Embedding(n_markets, embed_dim_market)
    self.sub_emb = nn.Embedding(n_submarkets, embed_dim_sub)

    # small bias terms (learned intercepts)
    self.market_intercept = nn.Embedding(n_markets, 1)
    self.sub_intercept = nn.Embedding(n_submarkets, 1)

    # market-level slope embedding (a scalar per market) for x1
    if use_market_varying_slope:
        self.market_slope = nn.Embedding(n_markets, 1)
    else:
        self.market_slope = None

    # MLP input dimension: x features + concatenated embeddings
    mlp_input_dim = 0
    # numeric inputs: x1, x2 (2)
    mlp_input_dim += 2
    # embedding dims
    mlp_input_dim += embed_dim_market + embed_dim_sub
    # optionally also include scalar market_slope so MLP can use it (not required)
    if use_market_varying_slope:
        mlp_input_dim += 1
    # build MLP
    layers = []
    in_dim = mlp_input_dim
    for h in mlp_hidden:
        layers.append(nn.Linear(in_dim, h))
        layers.append(nn.ReLU())
        in_dim = h
    # final output is a single scalar (prediction residual / non-linear contribution)
    layers.append(nn.Linear(in_dim, 1))
    self.mlp = nn.Sequential(*layers)

    # global intercept and global slope for x2 (learnable)
    self.global_intercept = nn.Parameter(torch.tensor([0.0], dtype=torch.float32))
    self.global_slope_x2 = nn.Parameter(torch.tensor([0.0], dtype=torch.float32))

    # initialization (small)
    nn.init.normal_(self.market_intercept.weight, mean=0.0, std=0.1)
    nn.init.normal_(self.sub_intercept.weight, mean=0.0, std=0.1)
    if use_market_varying_slope:
        nn.init.normal_(self.market_slope.weight, mean=0.0, std=0.1)

def forward(self, x1, x2, market_idx, sub_idx):
    """
    Inputs:
      x1, x2: tensors shape (batch, 1)
      market_idx, sub_idx: long tensors (batch,)
    Returns:
      y_hat (batch, 1)
    """
    # embeddings
    m_emb = self.market_emb(market_idx)  # (batch, embed_dim_market)
    s_emb = self.sub_emb(sub_idx)       # (batch, embed_dim_sub)

    # intercept contributions
    m_int = self.market_intercept(market_idx)  # (batch, 1)
    s_int = self.sub_intercept(sub_idx)       # (batch, 1)

    # market slope
    if self.use_market_varying_slope and self.market_slope is not None:
        m_slope = self.market_slope(market_idx)  # (batch, 1)
    else:
        m_slope = torch.zeros_like(x1)

    # linear part: global intercept + group intercepts + slope*x1 + global slope*x2
    linear_part = self.global_intercept + m_int + s_int + (m_slope * x1) + (self.global_slope_x2
* x2)

    # MLP part: feed numeric predictors and embeddings (optionally include m_slope scalar)
    mlp_in = torch.cat([x1, x2, m_emb, s_emb, m_slope], dim=1) if (self.use_market_varying_slope
and self.market_slope is not None) else torch.cat([x1, x2, m_emb, s_emb], dim=1)
    nonlin = self.mlp(mlp_in)  # (batch,1)

    y_hat = linear_part + nonlin
    return y_hat

def get_market_intercepts(self) -> np.ndarray:
    """Return learned market intercepts (as numpy array)."""
    return self.market_intercept.weight.detach().cpu().numpy().squeeze()

def get_submarket_intercepts(self) -> np.ndarray:
    return self.sub_intercept.weight.detach().cpu().numpy().squeeze()

def get_market_slope(self) -> Optional[np.ndarray]:
    if self.market_slope is None:
        return None
    return self.market_slope.weight.detach().cpu().numpy().squeeze()

def get_global_intercept_and_slope(self) -> Tuple[float, float]:
    return float(self.global_intercept.detach().cpu().item()),
float(self.global_slope_x2.detach().cpu().item())


-----------------------
```

Training utilities

```
def train_model(
model: nn.Module,
train_loader: DataLoader,
val_loader: Optional[DataLoader] = None,
lr: float = 1e-3,
epochs: int = 50,
weight_decay: float = 0.0,
device: torch.device = torch.device("cpu"),
):
optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=weight_decay)
criterion = nn.MSELoss()
model.to(device)
```

```
history = {"train_loss": [], "val_loss": []}
best_val = math.inf
best_state = None
for epoch in range(1, epochs + 1):
    model.train()
    running = 0.0
    count = 0
    for batch in train_loader:
        y = batch["y"].to(device)
        x1 = batch["x1"].to(device)
        x2 = batch["x2"].to(device)
        midx = batch["market_idx"].to(device)
        sidx = batch["submarket_idx"].to(device)

        optimizer.zero_grad()
        y_hat = model(x1, x2, midx, sidx)
        loss = criterion(y_hat, y)
        loss.backward()
        optimizer.step()

        running += loss.item() * y.size(0)
        count += y.size(0)

    train_loss = running / count
    history["train_loss"].append(train_loss)

    # validation
    if val_loader is not None:
        model.eval()
        vrunning = 0.0
        vcount = 0
        with torch.no_grad():
            for vb in val_loader:
                yv = vb["y"].to(device)
                xv1 = vb["x1"].to(device)
                xv2 = vb["x2"].to(device)
                vmidx = vb["market_idx"].to(device)
                vsidx = vb["submarket_idx"].to(device)
                yvhat = model(xv1, xv2, vmidx, vsidx)
                vloss = criterion(yvhat, yv)
                vrunning += vloss.item() * yv.size(0)
                vcount += yv.size(0)
        val_loss = vrunning / vcount
        history["val_loss"].append(val_loss)

        # save best
        if val_loss < best_val:
            best_val = val_loss
            best_state = {k: v.cpu().clone() for k, v in model.state_dict().items()}

        if epoch % 10 == 0 or epoch == 1:
            print(f"Epoch {epoch:3d} | train_loss: {train_loss:.4f} | val_loss: {val_loss:.4f}")
    else:
        if epoch % 10 == 0 or epoch == 1:
            print(f"Epoch {epoch:3d} | train_loss: {train_loss:.4f}")

# restore best weights if available
if best_state is not None:
    model.load_state_dict(best_state)
return history
```

def evaluate_model(model: nn.Module, loader: DataLoader, device: torch.device = torch.device("cpu")):

model.eval()

criterion = nn.MSELoss(reduction="mean")

running = 0.0

count = 0

with torch.no_grad():

for batch in loader:

y = batch["y"].to(device)

x1 = batch["x1"].to(device)

x2 = batch["x2"].to(device)

midx = batch["market_idx"].to(device)

sidx = batch["submarket_idx"].to(device)

yhat = model(x1, x2, midx, sidx)

loss = criterion(yhat, y)

running += loss.item() * y.size(0)

count += y.size(0)

return running / count

Utility: extract implied intercept for each market/submarket

----

def compute_group_intercepts_and_slopes(model: HierarchicalMLP, device: torch.device = torch.device("cpu")):

# To measure the net learned intercept for each market-submarket pair, we can:

# - set numeric predictors x1,x2 to zero

# - compute linear part (global_intercept + market_intercept + submarket_intercept)

# - add mlp contribution with x=0 (this depends on embeddings only)

model.eval()

with torch.no_grad():

n_markets = model.market_intercept.num_embeddings

n_subs = model.sub_intercept.num_embeddings

```
    # compute market-level intercept (global + market_emb + mlp contribution with zeros)
    market_idxs = torch.arange(n_markets, dtype=torch.long, device=device)
    sub_idxs_dummy = torch.zeros(n_markets, dtype=torch.long, device=device)  # just pick sub0 for ML input dims
    x1_zero = torch.zeros((n_markets, 1), dtype=torch.float32, device=device)
    x2_zero = torch.zeros((n_markets, 1), dtype=torch.float32, device=device)
    # get model components
    m_emb = model.market_emb(market_idxs)
    s_emb = model.sub_emb(sub_idxs_dummy)
    m_int = model.market_intercept(market_idxs)
    s_int = model.sub_intercept(sub_idxs_dummy)
    if model.use_market_varying_slope and model.market_slope is not None:
        m_slope = model.market_slope(market_idxs)
        mlp_in = torch.cat([x1_zero, x2_zero, m_emb, s_emb, m_slope], dim=1)
    else:
        m_slope = torch.zeros_like(x1_zero)
        mlp_in = torch.cat([x1_zero, x2_zero, m_emb, s_emb], dim=1)
    nonlin = model.mlp(mlp_in)
    market_implied_intercept = model.global_intercept + m_int + s_int + nonlin  # note s_int was
sub0's intercept
    market_implied_intercept = market_implied_intercept.squeeze().cpu().numpy()

    # for submarkets, compute implied intercept for each submarket by setting x=0 and using
corresponding embeddings
    sub_idxs = torch.arange(n_subs, dtype=torch.long, device=device)
    # compute parent market for each submarket by using embedding distances? We assume you pass
original mapping when needed.
    x1_zero_s = torch.zeros((n_subs, 1), dtype=torch.float32, device=device)
    x2_zero_s = torch.zeros((n_subs, 1), dtype=torch.float32, device=device)
    # To compute using correct market_intercepts, we need a mapping sub->market. The model
itself doesn't store mapping,
    # so user should provide mapping. We'll compute general sub intercept component (global +
sub intercept + mlp contribution using sub embedding)
    s_emb2 = model.sub_emb(sub_idxs)
    m_emb_dummy = torch.zeros_like(s_emb2[:, : model.market_emb.embedding_dim])  # dummy
    if model.use_market_varying_slope and model.market_slope is not None:
        dummy_m_slope = torch.zeros((n_subs, 1), dtype=torch.float32, device=device)
        mlp_in_s = torch.cat([x1_zero_s, x2_zero_s, m_emb_dummy, s_emb2, dummy_m_slope], dim=1)
    else:
        mlp_in_s = torch.cat([x1_zero_s, x2_zero_s, m_emb_dummy, s_emb2], dim=1)
    nonlin_s = model.mlp(mlp_in_s)
    sub_implied_intercept_component = model.global_intercept + s_int.squeeze()  # s_int was for
index 0, so not generic; better user provides mapping

# Note: computing exact implied intercept for each submarket needs the mapping sub->market.
# We return the raw learned embeddings & components so the user can combine them with their
mapping.
return {
    "market_intercept_learned": model.get_market_intercepts(),
    "submarket_intercept_learned": model.get_submarket_intercepts(),
    "market_slope_learned": model.get_market_slope(),
    "global_intercept": float(model.global_intercept.detach().cpu().item()),
    "global_slope_x2": float(model.global_slope_x2.detach().cpu().item()),
}
```

Example run (simulate + train)

```
def example_run():
# simulate data
df = simulate_hierarchical_data(n_markets=10, n_sub_per_market=5, n_per_submarket=80, include_varying_slope=True)
n_markets = int(df["market_idx"].nunique())
n_subs = int(df["submarket_idx"].nunique())
print("Data shape:", df.shape, "markets:", n_markets, "submarkets:", n_subs)
```

```
# train/test split (grouped by rows; this is a simple random split)
train_df, test_df = train_test_split(df, test_size=0.2, random_state=SEED)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
train_ds = HierDataset(train_df, device=device)
test_ds = HierDataset(test_df, device=device)
train_loader = DataLoader(train_ds, batch_size=512, shuffle=True)
test_loader = DataLoader(test_ds, batch_size=1024, shuffle=False)

# model
model = HierarchicalMLP(
    n_markets=n_markets,
    n_submarkets=n_subs,
    embed_dim_market=8,
    embed_dim_sub=6,
    mlp_hidden=(128, 64),
    use_market_varying_slope=True,
).to(device)

# train
history = train_model(model, train_loader, val_loader=test_loader, lr=1e-3, epochs=100,
device=device)
plt.plot(history["train_loss"], label="train")
plt.plot(history["val_loss"], label="val")
plt.yscale("log")
plt.legend()
plt.title("Training loss")
plt.show()

# evaluate
mse = evaluate_model(model, test_loader, device=device)
print("Test MSE:", mse)

# extract learned group-level params
groups = compute_group_intercepts_and_slopes(model, device=device)
print("Global intercept, global slope x2:", groups["global_intercept"],
groups["global_slope_x2"])
print("Market intercepts (first 10):", groups["market_intercept_learned"][:10])
print("Market slopes for x1 (first 10):", groups["market_slope_learned"][:10] if
groups["market_slope_learned"] is not None else None)

# If you simulated data with known true market slopes & intercepts,
# you can compare learned embeddings to true values (you have them in simulate function if
returned)
return model, df
```

if **name** == "**main**":

model, df = example_run()

print("Done.")

Notes & tips

- This ML model mixes **explicit linear group effects** (market/sub intercepts and market slope for `x1`) with a **non-linear MLP** that can learn interactions and residual structure. That combination replicates the semantics of (varying intercepts + varying slopes) while being much more flexible for non-linear data.
- **Interpretability**:
    - Use `model.get_market_intercepts()` and `model.get_market_slope()` to inspect learned group-level offsets and slopes.
    - The MLP non-linear component will contain additional group-dependent structure; you can probe it by setting predictors to specific values (e.g., x1=0, x2=0) and varying market/submarket indices to see how the network's non-linear contribution changes.
- **Extensions**:

  - Add L2 regularization (weight_decay) or embedding dropout to avoid overfitting.
  - If you have many groups (hundreds/thousands) increase embedding dimensions or use embedding regularization.
  - For Bayesian uncertainty over group effects, you can convert to a Bayesian NN (e.g., with Pyro or by Monte-Carlo dropout / ensembles), or fit an ensemble of this model to get empirical uncertainty.
- **Alternative non-Bayesian ML options**:
  - Gradient boosted trees (XGBoost / LightGBM) with group-level target encodings can also approximate group effects, but explicit embeddings + NN gives a cleaner, interpretable analog of multilevel models and supports smooth non-linear interactions.

If you want, I can:

- adapt the model to multiple varying-slope predictors,
- add training callbacks (early stopping), saving/loading, or
- produce a version that fits using `skorch` or `PyTorch Lightning` for cleaner training loops.

Which next step would you like?

Great . Well Done .
Now next chanllenge I want you to implement the above with ordinary least square model

Perfect 👍 This is a great next step.
You want to **replicate the hierarchical model using OLS (Ordinary Least Squares)**.

OLS by itself doesn't have built-in multilevel structure like PyMC or mixed models.
But we can **mimic hierarchical effects** using **dummy variables (fixed effects)** for each level in the hierarchy:

- **Varying intercepts** → Include dummy variables for markets and submarkets.
- **Varying slopes** → Interact predictors with these market/submarket dummies.

This way, OLS estimates separate intercepts and optionally separate slopes for each market/submarket.

Here's a clean Python implementation using `statsmodels` (so you see the regression output, not just predictions):

```
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf
```

Example synthetic dataset

```
np.random.seed(42)
```

```
n_markets = 10
n_submarkets = 5
n_obs = 2000
```

Generate hierarchy IDs

```
markets = np.random.choice([f"Market_{i}" for i in range(n_markets)], size=n_obs)
submarkets = [f"{m}_Sub{j}" for m in markets for j in [np.random.randint(0, n_submarkets)]]
```

Predictor and response

```
x = np.random.randn(n_obs)
```

True hierarchical structure: different intercepts + different slopes by market

```
market_effects = {f"Market_{i}": np.random.randn() for i in range(n_markets)}
submarket_effects = {sm: np.random.randn() * 0.5 for sm in set(submarkets)}
```

```
y = (
2.0 # global intercept
+ np.array([market_effects[m] for m in markets]) # market-level intercepts
+ np.array([submarket_effects[s] for s in submarkets]) # submarket-level intercepts
+ (1.5 + 0.5 * np.random.randn(n_obs)) * x # global slope + some noise
+ 0.5 * np.random.randn(n_obs) # residual noise
)
```

```
df = pd.DataFrame({
"y": y,
"x": x,
"market": markets,
"submarket": submarkets
})
```

Model 1: Varying intercepts only

```
model_intercepts = smf.ols("y ~ x + C(market) + C(submarket)", data=df).fit()
```

Model 2: Varying intercepts + varying slopes (by market)

Interaction term: predictor x market dummy

```
model_slopes = smf.ols("y ~ x * C(market) + C(submarket)", data=df).fit()

print("=== Varying intercepts only ===")
print(model_intercepts.summary())

print("\n=== Varying intercepts + varying slopes by market ===")
print(model_slopes.summary())
```

Explanation

- `C(market)` → creates dummy variables for each market (market-specific intercepts).
- `C(submarket)` → dummy variables for each submarket (nested intercepts).
- `x * C(market)` → allows slope of `x` to vary across markets (interaction).
- You can extend to **submarket-specific slopes** with `x * C(submarket)` if needed.

This is how OLS “fakes” a hierarchical model: instead of Bayesian pooling, it fits separate fixed effects for each level.

👉 Do you want me to also show how to **compare this OLS approach vs the hierarchical ML one** (e.g., out-of-sample prediction or parameter shrinkage differences)?

but waht about natinal level market ?

Good question — the **national level (United_state)** is already representable in OLS, but it works slightly differently than the market/submarket dummies. Below I'll:

1. explain how the national level maps to OLS coefficients,
2. show when you might add an explicit national dummy or national predictors, and
3. provide a clean code snippet that makes the national effect explicit and shows how to recover national → market → submarket decomposition.

Short answer:

- In an OLS regression the **global intercept** plays the role of the national-level intercept (`α_US`).
- `C(market)` and `C(submarket)` then estimate deviations from that national intercept (i.e., market and submarket offsets).
- If you want an explicit “national dummy” you don’t need one (there’s only one national level), but you can include national-level covariates (GDP, interest rate, season indicator, etc.) as additional

predictors — they'll estimate national effects on the outcome.

Below is a clean, copy-paste-ready code file that:

- simulates data for **`United_state -> markets -> submarkets`**,
- fits OLS with (a) national intercept + market & submarket fixed effects, and (b) adds varying slopes for **`x1`** by market, and (c) demonstrates how to extract **`alpha_US`**, market deviations, and submarket deviations so you can recombine them into group-level predictions.

"""

OLS-based hierarchical mimic: United_state -> Markets -> Submarkets

Features:

- National intercept (alpha_US) is the global intercept in OLS.
- Market and Submarket fixed effects are included as dummies (deviations from national).
- Optionally allow varying slopes for a predictor (x1) by market via interaction.
- Demonstrates how to extract alpha_US, market offsets, submarket offsets, and reconstruct group-level predictions.

Requirements:
pip install pandas numpy statsmodels

Run:
python ols_hierarchical_with_national_effect.py
"""

import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

np.random.seed(42)

def simulate_data(n_markets=10, n_sub_per_market=5, n_per_submarket=80):
"""Simulate data with a national intercept, market & submarket deviations, and market-varying slope for x1."""
n_submarkets = n_markets * n_sub_per_market
N = n_submarkets * n_per_submarket

```
# build hierarchical indices
market_idx = np.repeat(np.arange(n_markets), n_sub_per_market * n_per_submarket)
submarket_local = np.tile(np.repeat(np.arange(n_sub_per_market), n_per_submarket), n_markets)
submarket_global = np.concatenate([np.arange(m * n_sub_per_market, (m + 1) * n_sub_per_market)
for m in range(n_markets)])
submarket_idx = np.repeat(submarket_global, n_per_submarket)

# convert to labels (strings) to use categorical dummies easily in formulas
market_label = np.array([f"Market_{m}" for m in market_idx])
submarket_label = np.array([f"Market_{m}_Sub_{s}" for m, s in zip(market_idx, submarket_local)])

# predictors
x1 = np.random.normal(size=N)  # will have market-varying slope
x2 = np.random.normal(size=N)  # global fixed slope
x1 = (x1 - x1.mean()) / x1.std()
x2 = (x2 - x2.mean()) / x2.std()

# true generative params
alpha_US = 2.0  # national intercept
sigma_M_intercept = 0.8
```

```
sigma_M_slope = 0.5
rho = 0.25
sigma_S = 0.4
sigma_y = 1.0
beta_x2 = 0.6

# market-level correlated intercept & slope
cov_M = np.array([
    [sigma_M_intercept**2, rho * sigma_M_intercept * sigma_M_slope],
    [rho * sigma_M_intercept * sigma_M_slope, sigma_M_slope**2]
])
b_M = np.random.multivariate_normal(mean=[0.0, 0.0], cov=cov_M, size=n_markets)
market_intercept_true = b_M[:, 0]
market_slope_x1_true = b_M[:, 1]

# submarket-level intercepts
submarket_intercept_true = np.random.normal(0, sigma_S, size=n_submarkets)

# assemble
mu = (
    alpha_US
    + market_intercept_true[market_idx]
    + submarket_intercept_true[submarket_idx]
    + market_slope_x1_true[market_idx] * x1
    + beta_x2 * x2
)
y = mu + np.random.normal(0, sigma_y, size=N)

df = pd.DataFrame({
    "y": y,
    "x1": x1,
    "x2": x2,
    "market": market_label,
    "submarket": submarket_label
})
# Keep integer ids if needed
df["market_idx"] = market_idx
df["submarket_idx"] = submarket_idx
return df
```

```
def fit_ols_models(df):
"""
Fit two OLS models:
- model_intercepts: national intercept + market & submarket dummies + x1,x2 (intercepts-only)
- model_varying_slopes: national intercept + market & submarket dummies + x2 + x1*C(market)
(varying slopes by market)
"""
# Model A: varying intercepts only (national intercept is the overall intercept)
model_intercepts = smf.ols("y ~ x1 + x2 + C(market) + C(submarket)", data=df).fit()
```

```
# Model B: varying intercepts + varying slope for x1 by market (interact x1 with market dummies)
# Note: Using x1 * C(market) would include the global x1 effect plus market-specific
interactions;
# here we remove the global x1 to keep market-specific slopes only (you can keep global if
desired).
model_varying_slopes = smf.ols("y ~ x2 + C(market) + C(submarket) + x1:C(market)",
data=df).fit()

return model_intercepts, model_varying_slopes
```

```
def extract_group_effects(model, df):
"""
Extract alpha_US (national intercept), market offsets, and submarket offsets from a fitted model.
Returns:
alpha_US: float
market_offsets: dict market_label -> offset (deviation from alpha_US)
```

submarket_offsets: dict submarket_label -> offset (deviation from alpha_US + market offset)

Note: statsmodels parameterization uses one omitted category as reference (the intercept).

"""

params = model.params

# global intercept is the national-level intercept (alpha_US)

alpha_US = params.get("Intercept", 0.0)

```
# market offsets: parameters named like C(market)[T.Market_k]
market_offsets = {}
for name, val in params.items():
    if name.startswith("C(market)"):
        # name example: 'C(market)[T.Market_3]'
        market_label = name.split("]")[-1].strip() if "]" in name else name
        # alternative robust parse:
        # take substring between 'C(market)[T.' and ']'
        import re
        m = re.search(r"C\(market\)\[T\.(.+)\]", name)
        if m:
            market_label = m.group(1)
        market_offsets[market_label] = float(val)

# submarket offsets: C(submarket)[T.Market_0_Sub_3]
submarket_offsets = {}
for name, val in params.items():
    if name.startswith("C(submarket)"):
        import re
        m = re.search(r"C\(submarket\)\[T\.(.+)\]", name)
        if m:
            sub_label = m.group(1)
        else:
            sub_label = name
        submarket_offsets[sub_label] = float(val)

return alpha_US, market_offsets, submarket_offsets
```

def reconstruct_group_prediction(alpha_US, market_offsets, submarket_offsets, market_label, submarket_label, x1=0.0, x2=0.0, model=None):

"""

Reconstruct predicted mean for a given group (market/submarket) and predictors.

If model includes varying slopes via x1:C(market), model must be supplied to recover slope terms.

"""

pred = alpha_US

# add market offset if present

if market_label in market_offsets:

pred += market_offsets[market_label]

# add submarket offset if present

if submarket_label in submarket_offsets:

pred += submarket_offsets[submarket_label]

```
# add x2 global slope if available in model params
if model is not None and "x2" in model.params:
    pred += model.params["x2"] * x2

# add x1 market-specific slope if present
if model is not None:
    # find parameter x1:C(market)[T.Market_k]
    key = f"x1:C(market)[T.{market_label}]"
    if key in model.params:
        pred += model.params[key] * x1
    else:
        # maybe global x1 is present
        if "x1" in model.params:
            pred += model.params["x1"] * x1
```

```
return pred
```

def main():

df = simulate_data()

print("Simulated data:", df.shape)

model_intercepts, model_varying_slopes = fit_ols_models(df)

```
print("\n--- Model (intercepts only) summary (top) ---")
print(model_intercepts.summary().tables[0])  # header info
print("R-squared:", model_intercepts.rsquared)

print("\n--- Model (varying slopes) summary (top) ---")
print(model_varying_slopes.summary().tables[0])
print("R-squared:", model_varying_slopes.rsquared)

# Extract group effects from varying-slopes model
alpha_US, market_offsets, submarket_offsets = extract_group_effects(model_varying_slopes, df)
print("\nNational intercept (alpha_US):", alpha_US)
# show a few markets offsets
print("Market offsets (examples):", list(market_offsets.items())[:5])
print("Submarket offsets (examples):", list(submarket_offsets.items())[:5])

# Reconstruct prediction for a sample observation
sample = df.sample(1).iloc[0]
pred_manual = reconstruct_group_prediction(alpha_US, market_offsets, submarket_offsets,
                                          sample["market"], sample["submarket"],
                                          x1=sample["x1"], x2=sample["x2"],
model=model_varying_slopes)
pred_model = model_varying_slopes.predict(sample.to_frame().T)[0]
print("\nManual reconstructed prediction:", pred_manual)
print("Model prediction (verify):", pred_model)
```

if **name** == "**main**":

main()

Notes & guidance

- **National intercept (`alpha_US`)**: this is the `Intercept` reported by statsmodels and represents the baseline (United_state) mean when all dummies are in their reference categories and numeric predictors are zero (or at their mean if centered).
- **Market / Submarket offsets**: `C(market)[T.Market_k]` are deviations from the national intercept. If a market does not appear in the parameter list, it is the omitted reference category (offset = 0).
- **Varying slopes**: implementing `x1:C(market)` yields a coefficient `x1:C(market)[T.Market_k]` for each market (slope multiplier). If you want a global `x1` plus market adjustments, use `x1 * C(market)` instead.
- **Interpretation**: OLS with many dummies is equivalent to fitting a saturated fixed-effects model (no partial pooling). If you want partial pooling like in hierarchical Bayes / mixed models, prefer `mixedlm` (frequentist random effects) or Bayesian approaches.

If you want, next I can:

- show how to center predictors so `alpha_US` maps to the national mean more directly (e.g., set numeric predictors to zero at mean),
- show how to estimate the reference-category choice (change omitted category) to make a particular market the reference, or

- convert the OLS design into a **`patsy`** design matrix and show the exact columns and how to map them back to group-level terms.

> but i this model_intercepts = smf.ols("y ~ x + C(market) + C(submarket)", data=df).fit()
> where did you define that it is united_state level , How does model know that It is United_state level market ?

Ah — that's the subtle part 👍

In **OLS with dummies**, there's no explicit **`"United_state"`** category because there's only **one nation**. The **national-level effect is absorbed into the intercept**.

Let me break this carefully:

---

1. National level = global intercept

- In **`y ~ x + C(market) + C(submarket)`**
  - **`Intercept`** = **baseline national-level effect** (**`United_state`**).
  - **`C(market)`** = deviations of each market from that national baseline.
  - **`C(submarket)`** = deviations of each submarket from their market baseline (nested deviation).

So the model *implicitly* assumes a **root level (United_state)** through the intercept.
We don't need to code **`"United_state"`** as a category, because everything is under it.

---

2. How the model "knows" this

OLS design matrices are constructed with a **reference category**.

- Suppose we have 10 markets. Statsmodels will create 9 dummy variables (Market_1 … Market_9).
- The **missing one (Market_0)** is the reference → its effect is absorbed in the intercept.
- That intercept is interpreted as: **United_state baseline + Market_0 effect (reference market) + Submarket_0 effect (reference submarket)**.

So, "United_state" is *not* a column in your dataset, it's the **root effect** captured by the global intercept.

---

3. If you want to make "United_state" explicit

You could add a dummy column like **`is_united_state = 1`** for every row. But that's redundant, because the intercept already plays that role.

For clarity, you can **center predictors** so that:

- Intercept (**`alpha_US`**) = national mean across all markets/submarkets.
  That way you can directly say: *this intercept is the United_state-level average outcome when x = 0*.

✅ So:

- You don't see `"United_state"` in the formula.
- But it's always there — it's the **intercept**.
- Market and submarket dummies are deviations from that national baseline.

Would you like me to **rewrite the OLS formula** so that the intercept *exactly* corresponds to the **United_state national average**, instead of being tied to the reference market/submarket? (We can do this with effects coding or grand mean centering.)

> can you show how you have create a dataset for that ?

Perfect — let's go step by step and **explicitly build a dataset with 3 levels**:

- **United_state** (national level, 1 root node)
- **10 Markets** (children of United_state)
- **5 Submarkets per Market** (children of each market)

We'll simulate predictors and outcomes so that:

- There is a **national-level intercept** (baseline for the whole US).
- Each **market** has its own deviation.
- Each **submarket** has its own deviation.
- Optionally, predictors (`x1`, `x2`) have effects that may vary across levels.

Here's a clean Python example:

```
import numpy as np
import pandas as pd

np.random.seed(42)
```

Hierarchy sizes

```
n_markets = 10
n_submarkets = 5
n_obs = 100 # per submarket
```

National-level intercept (United_state)

```
alpha_us = 5.0
```

Market- and submarket-level deviations

market_effects = np.random.normal(0, 1.0, n_markets)
submarket_effects = np.random.normal(0, 0.5, n_markets * n_submarkets)

Predictor effects (fixed globally)

beta_x1 = 2.0
beta_x2 = -1.0

Generate dataset

records = []
for m in range(n_markets):
for s in range(n_submarkets):
for i in range(n_obs):
# Predictors
x1 = np.random.randn()
x2 = np.random.randn()

```
        # Outcome: national + market + submarket + predictors + noise
        mu = (alpha_us
              + market_effects[m]
              + submarket_effects[m * n_submarkets + s]
              + beta_x1 * x1
              + beta_x2 * x2)
        y = mu + np.random.normal(0, 1.0)  # residual

        records.append({
            "United_state": "US",          # National root
            "market": f"Market_{m}",       # Market ID
            "submarket": f"Market_{m}_Sub{s}",  # Submarket ID
            "x1": x1,
            "x2": x2,
            "y": y
        })
```

df = pd.DataFrame.from_records(records)
print(df.head(10))
print("Shape:", df.shape)

Output (example head)

```python-repl
  United_state    market       submarket        x1        x2           y
0          US  Market_0  Market_0_Sub0  -0.138264  0.647689   6.123
1          US  Market_0  Market_0_Sub0   1.523030 -0.234137  12.547
2          US  Market_0  Market_0_Sub0  -0.234153 -0.234137   4.879
3          US  Market_0  Market_0_Sub0   0.767435 -0.469474   8.024
...
Shape: (5000, 6)
```

Key points:

- The `United_state` column is **constant** (“US”), since there’s only one national node.
- The **national effect (`alpha_us`)** is part of the data-generating process (baseline = 5.0).

- Market effects (`market_effects`) are deviations from the national intercept.
- Submarket effects (`submarket_effects`) are deviations within markets.
- Predictors (`x1`, `x2`) apply across all levels.

👉 This way, you can later fit `OLS` like:

```
python

import statsmodels.formula.api as smf

model = smf.ols("y ~ x1 + x2 + C(market) + C(submarket)", data=df).fit()
print(model.summary())
```

Here the **Intercept ≈ national effect (United_state)**, while `C(market)` and `C(submarket)` estimate deviations.

Do you want me to also show how to **reconstruct predictions** as
`y_hat = α_US + α_market + α_submarket + βx`
from the OLS coefficients, so the three-level decomposition is explicit?

Now next most difficult challenge you have is , can you create the above model with lightgbm ?

Great 👌 now we're moving into **nonlinear ML territory** with LightGBM.

Unlike OLS or PyMC, LightGBM doesn't know about "hierarchical intercepts" automatically — but we can **teach it the hierarchy** by feeding it **categorical features** (`market`, `submarket`, and even `United_state`).

Here's a clean implementation using your dataset generator + LightGBM:

---

import numpy as np
import pandas as pd
import lightgbm as lgb
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

Step 1: Generate dataset

----

np.random.seed(42)

n_markets = 10
n_submarkets = 5
n_obs = 100

```
alpha_us = 5.0
market_effects = np.random.normal(0, 1.0, n_markets)
submarket_effects = np.random.normal(0, 0.5, n_markets * n_submarkets)
```

```
beta_x1 = 2.0
beta_x2 = -1.0
```

```
records = []
for m in range(n_markets):
for s in range(n_submarkets):
for i in range(n_obs):
x1 = np.random.randn()
x2 = np.random.randn()
mu = (alpha_us
+ market_effects[m]
+ submarket_effects[m * n_submarkets + s]
+ beta_x1 * x1
+ beta_x2 * x2)
y = mu + np.random.normal(0, 1.0)
records.append({
"United_state": "US",
"market": f"Market_{m}",
"submarket": f"Market_{m}_Sub{s}",
"x1": x1,
"x2": x2,
"y": y
})
```

```
df = pd.DataFrame.from_records(records)
```

Step 2: Train-test split

```
X = df[["United_state", "market", "submarket", "x1", "x2"]]
y = df["y"]
```

Encode categorical features as category dtype

```
X["United_state"] = X["United_state"].astype("category")
X["market"] = X["market"].astype("category")
X["submarket"] = X["submarket"].astype("category")
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step 3: Train LightGBM

```
train_data = lgb.Dataset(X_train, label=y_train, categorical_feature=["United_state", "market",
"submarket"])
test_data = lgb.Dataset(X_test, label=y_test, categorical_feature=["United_state", "market",
"submarket"], reference=train_data)

params = {
"objective": "regression",
"metric": "rmse",
"learning_rate": 0.1,
"num_leaves": 31,
"feature_fraction": 0.8,
"bagging_fraction": 0.8,
"bagging_freq": 5,
"verbose": -1
}

model = lgb.train(params, train_data, valid_sets=[train_data, test_data], num_boost_round=500,
early_stopping_rounds=50)
```

Step 4: Evaluate

```
y_pred = model.predict(X_test, num_iteration=model.best_iteration)
rmse = mean_squared_error(y_test, y_pred, squared=False)
print("RMSE:", rmse)
```

Step 5: Feature importance

```
importance = pd.DataFrame({
"feature": model.feature_name(),
"importance": model.feature_importance()
}).sort_values(by="importance", ascending=False)

print("\nFeature Importances:")
print(importance)
```

🔑 Key ideas

1. **Hierarchy encoding**
    - `United_state`, `market`, and `submarket` are passed as categorical features.
    - LightGBM learns splits like:
        - *If market = Market_3 → adjust predictions accordingly*
        - *If submarket = Market_7_Sub2 → adjust further*.
2. **National level**
    - `United_state` is redundant here (only one value = "US"). But including it keeps the 3-level structure explicit.
    - Effectively, the **baseline prediction** corresponds to `United_state`.
3. **Nonlinear capability**
    - Unlike OLS, LightGBM doesn't assume linear slopes.
    - It can capture nonlinear relationships between predictors (`x1`, `x2`) and the target.

👉 Next step (if you want): I can show you how to **replicate varying slopes** in LightGBM — by letting it interact predictors with markets/submarkets automatically. Would you like me to demonstrate that?

Sure

Perfect! 💪 Let's extend the LightGBM model so it can **learn varying slopes**, i.e., the effect of `x1` can **change per market or submarket**.

In OLS we did this with `x1 * C(market)`. In LightGBM, we mimic this with **interaction features**. LightGBM will then treat these interactions as separate features and learn **market-specific slopes**.

Here's a clean example:

```
import numpy as np
import pandas as pd
import lightgbm as lgb
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

np.random.seed(42)
```

Step 1: Generate hierarchical dataset

```
n_markets = 10
n_submarkets = 5
n_obs = 100

alpha_us = 5.0
market_effects = np.random.normal(0, 1.0, n_markets)
submarket_effects = np.random.normal(0, 0.5, n_markets * n_submarkets)

beta_x1 = 2.0
beta_x2 = -1.0

records = []
for m in range(n_markets):
for s in range(n_submarkets):
for i in range(n_obs):
x1 = np.random.randn()
x2 = np.random.randn()
mu = (alpha_us
+ market_effects[m]
+ submarket_effects[m * n_submarkets + s]
+ beta_x1 * x1
+ beta_x2 * x2)
y = mu + np.random.normal(0, 1.0)
records.append({
"United_state": "US",
"market": f"Market_{m}",
"submarket": f"Market_{m}_Sub{s}",
"x1": x1,
"x2": x2,
"y": y
})

df = pd.DataFrame.from_records(records)
```

Step 2: Create interaction features for varying slopes

```
for m in range(n_markets):
df[f"x1_Market_{m}"] = df["x1"] * (df["market"] == f"Market_{m}").astype(int)
```

You can also create submarket interactions if needed:

for s in df["submarket"].unique():

df[f"x1_{s}"] = df["x1"] * (df["submarket"] == s).astype(int)

Step 3: Split train/test

```
feature_cols = ["x2"] + [f"x1_Market_{m}" for m in range(n_markets)]
X = df[feature_cols]
y = df["y"]
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step 4: Train LightGBM

```
train_data = lgb.Dataset(X_train, label=y_train)
test_data = lgb.Dataset(X_test, label=y_test, reference=train_data)
```

```
params = {
"objective": "regression",
"metric": "rmse",
"learning_rate": 0.1,
"num_leaves": 31,
"verbose": -1
}
```

```
model = lgb.train(params, train_data, valid_sets=[train_data, test_data],
num_boost_round=500, early_stopping_rounds=50)
```

Step 5: Evaluate

```
y_pred = model.predict(X_test, num_iteration=model.best_iteration)
rmse = mean_squared_error(y_test, y_pred, squared=False)
print("RMSE:", rmse)
```

Feature importance

```
importance = pd.DataFrame({
"feature": model.feature_name(),
"importance": model.feature_importance()
}).sort_values(by="importance", ascending=False)

print("\nFeature importances:")
print(importance)
```

✅ How it works

1. **Interactions for varying slopes**
   - `x1_Market_0 = x1 * (market == Market_0)`
   - LightGBM sees this as a **separate feature**, so the slope of `x1` can vary by market.
   - You can do the same for submarkets if needed.
2. **Global baseline**
   - Even without `United_state` as a feature, the model learns a **global baseline** via the boosting trees.
3. **Advantages over OLS**
   - Captures **nonlinear interactions** automatically.
   - No need to manually code complex dummy-variable interactions beyond `x1 * market`.

If you want, I can **also show a fully “nested hierarchical” LightGBM setup** where both **market and submarket varying slopes** are included automatically, **without manually creating hundreds of features** — using category encoding and interaction tricks.

Do you want me to do that next?

Sure

Perfect! 💪 Let’s build a **fully nested hierarchical LightGBM model** with both **market- and submarket-varying slopes**, **without manually creating hundreds of features**.

The idea:

1. Treat `market` and `submarket` as **categorical features**.
2. Use **LightGBM categorical handling** to let the model split on markets/submarkets.
3. Encode **interactions automatically** by creating **interaction features** using `x1` and `x2` with category indices.

Here’s a clean, scalable implementation:

```
import numpy as np
import pandas as pd
import lightgbm as lgb
```

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
np.random.seed(42)
```

Step 1: Simulate hierarchical dataset

```
n_markets = 10
n_submarkets = 5
n_obs = 100
alpha_us = 5.0
market_effects = np.random.normal(0, 1.0, n_markets)
submarket_effects = np.random.normal(0, 0.5, n_markets * n_submarkets)
beta_x1 = 2.0
beta_x2 = -1.0
records = []
for m in range(n_markets):
for s in range(n_submarkets):
for i in range(n_obs):
x1 = np.random.randn()
x2 = np.random.randn()
mu = (alpha_us
+ market_effects[m]
+ submarket_effects[m * n_submarkets + s]
+ beta_x1 * x1
+ beta_x2 * x2)
y = mu + np.random.normal(0, 1.0)
records.append({
"United_state": "US",
"market": f"Market_{m}",
"submarket": f"Market_{m}_Sub{s}",
"x1": x1,
"x2": x2,
"y": y
})
df = pd.DataFrame.from_records(records)
```

Step 2: Encode hierarchical categorical features

LightGBM can handle categorical features natively

```
cat_features = ["United_state", "market", "submarket"]
for c in cat_features:
df[c] = df[c].astype("category")
```

Step 3: Create interaction features for predictors x1 and x2 with categories

----

This allows LightGBM to learn varying slopes

```
for c in ["market", "submarket"]:
df[f"x1_{c}"] = df["x1"].astype(float) * df[c].cat.codes.astype(float)
df[f"x2_{c}"] = df["x2"].astype(float) * df[c].cat.codes.astype(float)
```

Step 4: Train-test split

```
feature_cols = ["x1", "x2"] + [f"x1_{c}" for c in ["market","submarket"]] + [f"x2_{c}" for c in ["market","submarket"]] + cat_features
X = df[feature_cols]
y = df["y"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step 5: Train LightGBM

```
train_data = lgb.Dataset(X_train, label=y_train, categorical_feature=cat_features)
test_data = lgb.Dataset(X_test, label=y_test, reference=train_data, categorical_feature=cat_features)

params = {
"objective": "regression",
"metric": "rmse",
"learning_rate": 0.05,
"num_leaves": 31,
"feature_fraction": 0.8,
"bagging_fraction": 0.8,
"bagging_freq": 5,
"verbose": -1
}
```

```
model = lgb.train(params, train_data, valid_sets=[train_data, test_data],
num_boost_round=1000, early_stopping_rounds=50)
```

Step 6: Evaluate

```
y_pred = model.predict(X_test, num_iteration=model.best_iteration)
rmse = mean_squared_error(y_test, y_pred, squared=False)
print("RMSE:", rmse)
```

Step 7: Feature importance

```
importance = pd.DataFrame({
"feature": model.feature_name(),
"importance": model.feature_importance()
}).sort_values(by="importance", ascending=False)
print("\nFeature Importances:")
print(importance)
```

🔑 Key points

1. **National intercept**:
    - `United_state` is categorical, only 1 value ("US"). LightGBM uses it to anchor the baseline prediction.
2. **Market- and submarket-varying intercepts**:
    - `market` and `submarket` are categorical features → LightGBM can split on these to adjust predictions.
3. **Varying slopes**:

  - Features like `x1_market = x1 * market_code` allow **market-specific slopes**.
  - Features like `x1_submarket = x1 * submarket_code` allow **submarket-specific slopes**.

4. **Nonlinear interactions**:
   - LightGBM automatically models nonlinearities, so this setup **replicates hierarchical linear models** while allowing **nonlinear effects**.

If you want, I can next **draw a diagram showing how LightGBM splits mimic hierarchical intercepts and slopes**, so it's easier to visualize how this tree-based ML model replicates your 3-level hierarchy.

Do you want me to do that?

Sure

Perfect! Let's visualize how LightGBM mimics your **3-level hierarchical model** (United_state → market → submarket) and also allows **varying slopes**.

1️⃣ The hierarchy in your data

```python-repl
United_state (national level)
│
├─ Market_0
│    ├─ Submarket_0
│    ├─ Submarket_1
│    └─ Submarket_4
├─ Market_1
│    ├─ Submarket_0
│    ├─ Submarket_1
│    └─ Submarket_4
│
...
└─ Market_9
     ├─ Submarket_0
     ├─ Submarket_1
     └─ Submarket_4
```

- National intercept = **baseline prediction**
- Market intercepts = **deviations learned per market**
- Submarket intercepts = **further deviations**

2️⃣ How LightGBM learns this hierarchy

- **Step 1: Split on market** → adjusts the baseline prediction per market
- **Step 2: Split on submarket within a market** → further adjusts prediction
- **Step 3: Split on x1/x2 or x1*market, x1*submarket** → allows **varying slopes**

Example tree structure (simplified):

```yaml
[Root: prediction = alpha_US]
├─ if market == Market_3:  # Market-level intercept
│   ├─ if submarket == Market_3_Sub_2:  # Submarket-level intercept
│   │   ├─ if x1_Market_3 > 0.5:  # Market-specific slope for x1
│   │   │   predict += slope_adjustment
│   │   └─ else:
│   │       predict += slope_adjustment
│   └─ else:
│       predict += submarket_other_adjustment
└─ else:
    predict += market_other_adjustment
```

- **Root node** = **`United_state`** national baseline
- **First splits** = markets → varying intercepts
- **Next splits** = submarkets → nested varying intercepts
- **Splits on x1*market / x1*submarket** → varying slopes

LightGBM **doesn't need explicit formulas** like OLS; it automatically learns **intercept adjustments + slope interactions** via tree splits.

3️⃣ Visual intuition

```sql
Alpha_US (national baseline)
|
+-- Market_0 offset
|   +-- Submarket_0 offset + slope adjustment
|   +-- Submarket_1 offset + slope adjustment
|   ...
|
+-- Market_1 offset
|   +-- Submarket_0 offset + slope adjustment
|   ...
```

- Prediction = **`Alpha_US + Market_offset + Submarket_offset + slope_effects`**
- The slope effects depend on **interaction features**, e.g., **`x1 * market_code`**.
- Because trees can split multiple times, **nonlinear relationships** are automatically captured.

✅ Summary:

- Root node = national intercept
- Market splits = market-level intercepts
- Submarket splits = submarket-level intercepts
- Interaction features = varying slopes
- Tree structure + boosting = nonlinear hierarchical model

If you want, I can **draw an actual LightGBM decision tree diagram from your trained model** showing how it splits on market/submarket/x1 — that would make the hierarchy and varying slopes **visually obvious**.

Do you want me to do that?

Sure

Perfect! Let’s generate a **visual diagram of a LightGBM tree** from your hierarchical dataset so you can see exactly how it learns national → market → submarket intercepts and varying slopes.

Here’s a step-by-step Python example using your existing LightGBM model:

import lightgbm as lgb
import matplotlib.pyplot as plt

Assuming you already have a trained LightGBM model called `model`

Step 1: Plot the first tree

LightGBM has many trees; plotting the first tree shows the basic hierarchy learned

ax = lgb.plot_tree(model, tree_index=0, figsize=(20, 10), show_info=["split_gain", "internal_value", "internal_count", "leaf_count"])
plt.title("LightGBM Tree 0: Hierarchical Splits Visualization")
plt.show()

Step 2 (optional): Plot multiple trees

----

lgb.plot_tree(model, tree_index=1, figsize=(20,10)) # second tree

lgb.plot_tree(model, tree_index=2, figsize=(20,10)) # third tree

----

Step 3: Save tree to file (optional)

----

lgb.plot_tree(model, tree_index=0, figsize=(20,10))

plt.savefig("tree0.png")

🔑 Notes

1. **Root node**
    - Represents the **national-level baseline (United_state)**.
2. **First splits**
    - Often on `market` categorical features → captures **market-specific intercepts**.
3. **Second splits**
    - On `submarket` or interaction features like `x1_market` → captures **submarket-specific intercepts and varying slopes**.
4. **Leaves**
    - Each leaf represents a **prediction adjustment**: baseline + market + submarket + slope effect.
5. **Multiple trees**
    - LightGBM uses **many trees**; each tree adds incremental adjustments to refine the prediction.
    - The first few trees usually capture **intercepts (hierarchy)**, later trees refine **slopes and nonlinear effects**.

If you want, I can also **highlight exactly which nodes correspond to “market intercepts” vs “submarket intercepts” vs “x1 slope adjustments”** in the diagram, so you get a **fully annotated hierarchical view**.

Do you want me to do that?

Sure

Perfect! Let’s annotate a LightGBM tree so you can **see exactly which splits correspond to national, market, submarket, and varying slopes**.

Here’s a Python example using **matplotlib** and LightGBM’s `plot_tree` functionality, with **custom labels** for clarity:

```
import lightgbm as lgb
import matplotlib.pyplot as plt
```

Assuming you already have a trained LightGBM model called `model`

Step 1: Plot the first tree

```
ax = lgb.plot_tree(
model,
tree_index=0,
figsize=(20, 12),
show_info=["split_gain", "internal_value", "internal_count", "leaf_count"]
)
plt.title("LightGBM Tree 0: Hierarchical Splits with Annotations")
```

Step 2: Annotate the splits

Get the booster

```
booster = model.booster_
tree = booster.dump_model()["tree_info"][0]["tree_structure"]

def annotate_node(node, depth=0):
if "split_index" in node:
feature = node["split_feature"]
feature_name = booster.feature_name()[feature]
# Determine type of effect
if feature_name == "United_state":
effect_type = "National intercept"
elif feature_name.startswith("market"):
effect_type = "Market intercept / slope"
elif feature_name.startswith("submarket"):
effect_type = "Submarket intercept / slope"
```

elif "x1" in feature_name or "x2" in feature_name:
effect_type = "Varying slope"
else:
effect_type = "Other predictor"

```
    print("  " * depth + f"Split: {feature_name} → {effect_type}")
    # Recurse
    annotate_node(node["left_child"], depth + 1)
    annotate_node(node["right_child"], depth + 1)
else:
    # Leaf node
    print("  " * depth + f"Leaf: value = {node['leaf_value']:.3f}")
```

Print annotated structure to console

print("\n=== Annotated LightGBM Tree 0 ===")
annotate_node(tree)

plt.show()

🔑 What this does

1. **Plots the first tree**
    - Shows hierarchical splits visually.
2. **Prints annotations in the console**
    - `"National intercept"` → usually the root node (`United_state`).
    - `"Market intercept / slope"` → splits on `market` or `x1_market`.
    - `"Submarket intercept / slope"` → splits on `submarket` or `x1_submarket`.
    - `"Varying slope"` → splits on `x1`, `x2`, or interaction features.
    - `"Leaf"` → final prediction adjustment at that node.
3. **Interpretation**
    - Root node = baseline national-level prediction
    - Market splits = market-level adjustments
    - Submarket splits = nested submarket adjustments
    - Interaction features = varying slopes per group

This way, you can **directly see how LightGBM approximates a hierarchical model**, including **national → market → submarket intercepts and varying slopes**, while also allowing **nonlinear adjustments**.

If you want, I can **combine the visual tree with colored boxes** so that **national, market, submarket, and slope nodes are color-coded**, making it even easier to interpret at a glance.

Do you want me to do that next?

Sure

Perfect! Let’s create a **color-coded LightGBM tree plot** where each type of node is visually distinct:

- **National intercept** → root node (e.g., blue)
- **Market intercept / slope** → green
- **Submarket intercept / slope** → orange
- **Varying slopes / predictors** → red
- **Leaf nodes** → gray

Here’s a complete example:

```
import lightgbm as lgb
import matplotlib.pyplot as plt
```

Assuming you already have a trained LightGBM model called `model`

```
booster = model.booster_
```

Step 1: Plot the first tree

```
ax = lgb.plot_tree(
model,
tree_index=0,
figsize=(24, 14),
show_info=["split_gain", "internal_value", "internal_count", "leaf_count"]
)
plt.title("Color-coded LightGBM Tree: Hierarchical Intercepts & Varying Slopes")
```

Step 2: Annotate and color nodes

LightGBM doesn’t provide direct node coloring in plot_tree,

so we adjust colors by traversing the tree and using matplotlib patches.

```
from matplotlib.patches import Rectangle
```

Helper function to recursively traverse tree and annotate colors

```
def traverse_and_color(node, ax, depth=0):
if "split_index" in node:
feature_idx = node["split_feature"]
feature_name = booster.feature_name()[feature_idx]
# Choose color
if feature_name == "United_state":
color = "lightblue"
elif feature_name.startswith("market"):
color = "lightgreen"
elif feature_name.startswith("submarket"):
color = "orange"
elif "x1" in feature_name or "x2" in feature_name:
color = "salmon"
else:
color = "white"
# Add annotation (text)
x = node.get("internal_count", 0) # approximate x
y = -depth # approximate y
ax.add_patch(Rectangle((x-0.5, y-0.5), 1, 1, facecolor=color, alpha=0.3))
ax.text(x, y, feature_name, ha="center", va="center", fontsize=8)
# Recurse
traverse_and_color(node["left_child"], ax, depth+1)
traverse_and_color(node["right_child"], ax, depth+1)
else:
# Leaf node
color = "lightgray"
x = node.get("internal_count", 0)
y = -depth
ax.add_patch(Rectangle((x-0.5, y-0.5), 1, 1, facecolor=color, alpha=0.3))
ax.text(x, y, f"{node['leaf_value']:.2f}", ha="center", va="center", fontsize=8)
```

Get tree structure

```
tree0 = booster.dump_model()["tree_info"][0]["tree_structure"]
traverse_and_color(tree0, ax)
```

```
plt.show()
```

🔑 How this works

1. **Color scheme**

Node type	Color
National intercept (root)	light blue
Market intercept / slope	light green
Submarket intercept / slope	orange
Varying slopes (x1/x2)	salmon
Leaf nodes	light gray

2. **Tree interpretation**

- **Blue root** → national baseline (`United_state`)
- **Green nodes** → market-level adjustments
- **Orange nodes** → submarket adjustments
- **Red/salmon nodes** → varying slopes (interactions with predictors)
- **Gray leaves** → final prediction increments

3. **Benefit**

- At a glance, you can see **how LightGBM splits mimic hierarchical intercepts and varying slopes**, while also allowing nonlinear effects.

If you want, I can **also create a compact diagram showing only the top hierarchy splits** (national → market → submarket) **without all internal slope splits**, which is easier to visualize for presentations.

Do you want me to do that?

Sure

Perfect! Let's create a **compact hierarchical diagram** showing only the **national → market → submarket splits**, ignoring the lower-level slope and nonlinear splits. This makes it clean and presentation-ready.

Here's a Python example using **networkx** for the hierarchy and **matplotlib** for visualization:

```
import networkx as nx
import matplotlib.pyplot as plt
```

Step 1: Build hierarchy

----

n_markets = 10
n_submarkets = 5

G = nx.DiGraph()

Root node: United_state

G.add_node("United_state")

Add markets

for m in range(n_markets):
market_name = f"Market_{m}"
G.add_node(market_name)
G.add_edge("United_state", market_name)

```
# Add submarkets
for s in range(n_submarkets):
    submarket_name = f"{market_name}_Sub{s}"
    G.add_node(submarket_name)
    G.add_edge(market_name, submarket_name)
```

Step 2: Plot hierarchy

----

plt.figure(figsize=(14, 8))

Use hierarchy layout

pos = nx.nx_agraph.graphviz_layout(G, prog="dot") # Requires pygraphviz or pydot

Draw nodes

nx.draw(G, pos, with_labels=True, node_size=2000, node_color="lightblue", font_size=10, arrows=True)

plt.title("Compact Hierarchical Structure: United_state → Market → Submarket")
plt.show()

🔑 What this diagram shows

1. **Root node** → `United_state` (national-level intercept)
2. **Second level** → Markets (`Market_0` … `Market_9`)

3. **Third level** → Submarkets (`Market_i_Sub_j`)
4. **No predictor splits shown** → clean hierarchy only
5. **Use in presentations** → communicates the hierarchical structure clearly

---

If you want, I can also **combine this compact hierarchy with LightGBM predictions** at each node, so each node shows:

- National baseline
- Market-specific offset
- Submarket-specific offset

This would make it a **fully annotated hierarchical prediction diagram**.

Do you want me to do that next?