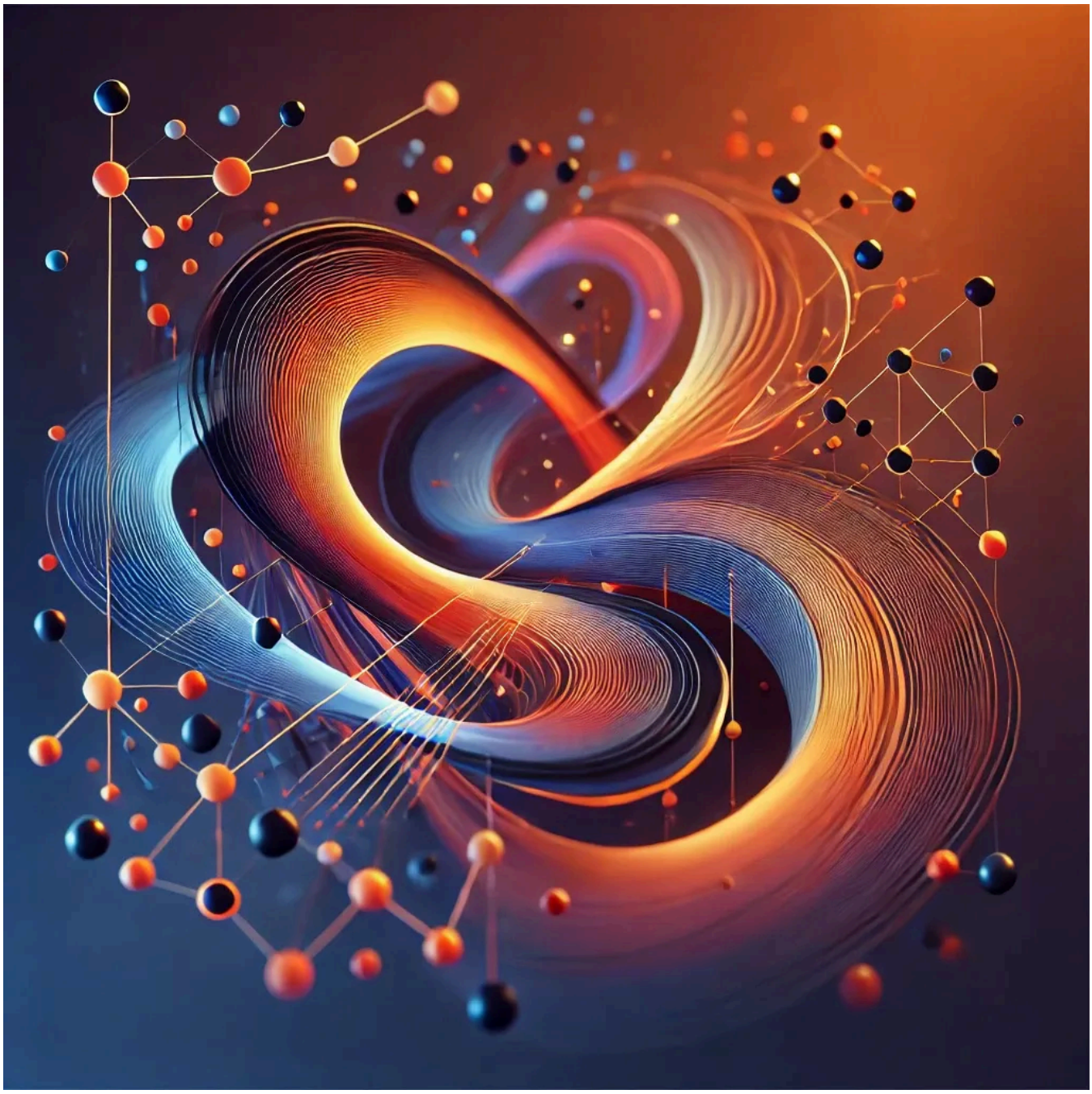
Bayesian Modeling with PYMC: Building Intuitive and Powerful Models

Uncover how to construct a Bayesian regression model with PYMC, complete with priors, likelihoods, and insightful design principles.

8 min read · Nov 24, 2024

J Justin Wall Follow



Bayesian Modeling Series using PYMC

Welcome to part one of a two-part series on Bayesian regression models with PYMC! In this first part, we'll introduce our dataset and walk through the process of building a Bayesian regression model using PYMC. Our focus will be on understanding the syntax, tools, and flexibility this powerful Python library offers. In part two, we'll shift gears to diagnostics and results, learning how to evaluate our model and interpret the feature coefficients in a meaningful way. When building a statistical model, we often focus on prediction. However, it's important to note that the goal of this series is to explore PYMC syntax and gain insights into the feature relationships to tell the story behind the data.

Here's what to expect:

- **Part 1:** *Bayesian Modeling with PYMC: Building Intuitive and Powerful Models*
- **Part 2:** *Interpreting Bayesian Models: Diagnostics and Insights with PYMC*

Let's dive in and start building!

Project Goal

The primary goal of this project is to explore and deeply understand the syntax of PYMC regression modeling. To keep things simple and focused, we've chosen the California Housing dataset as our working example. This article isn't about mastering prediction or uncovering the secrets of California's housing market — it's about learning the mechanics of Bayesian regression with PYMC.
To start, we'll walk through four key steps to prepare the dataset and build a regression model using Bayesian statistics. For installation instructions and detailed documentation, visit the the PYMC website.
Step 1: Define the problem
Step 2: Explore the California Housing Dataset
Step 3: Prepare Data for Regression
Step 4: Build PYMC Model & Learn the Syntax

What's Next?

In part two, we'll cover steps five through seven: diagnostics and results. Stay tuned!

Step 1: Define the Problem

A well-framed project statement and clear goal are essential for improving efficiency, delivering meaningful outcomes, and providing value to clients and stakeholders. Every data science project benefits from a thorough and specific problem statement to ensure it stays on track and achieves its purpose.
For this project, our goal is to use the California Housing dataset from the Scikit-Learn library to train a Bayesian regression model using PYMC. The target variable is the median house value for California districts. The focus is on interpreting the coefficients to understand how various features influence housing prices, rather than solely optimizing predictive performance.

Back to top

Step 2: Explore the California Housing Dataset

In a typical data science project, exploring and understanding the dataset is a time-intensive and crucial step. However, for this project, our focus is on PYMC and Bayesian regression, so we'll briefly review the dataset before diving into model building. Let's cover the basics with a quick assessment.

Initial Observations

From the initial data output, we see that all features are non-null and in a `float` data type. This simplifies preprocessing and allows us to focus on modeling.

Target Variable Distribution

The distribution of the target variable, median house value, appears relatively clean. While there is a slight right skew for higher-priced housing, most values cluster around a single peak with minimal outliers or gaps.

Addressing Skewness

To handle the skewed tail, we'll standardize the target variable in the next section. Standardization can help improve model performance and make the results easier to interpret. Alternatively, we could consider using a gamma distribution for the likelihood to better fit the skewness. However, for simplicity, we'll stick to standardization for this example.

Feature Definitions

Let's take a closer look at the features in the California Housing dataset and what they represent:

- **`MedHouseVal`**
  The median house value for California districts, expressed in hundreds of thousands of dollars. This is our target variable.

- **`MedInc`**
  Median income of a block group, where a block group is the smallest geographical unit for which the U.S. Census Bureau publishes sample data.

- **`HouseAge`**

  Median house age in the block group, expressed in years.

- **`AveRooms`**

  The average number of rooms per household in the block group.

- **`AveBedrms`**

  The average number of bedrooms per household in the block group.

- **`Population`**

  The total population of the block group.

- **`AveOccup`**

  The average number of household members per household in the block group.

- **`Latitude`**

  The geographical latitude of the block group.

- **`Longitude`**

  The geographical longitude of the block group.

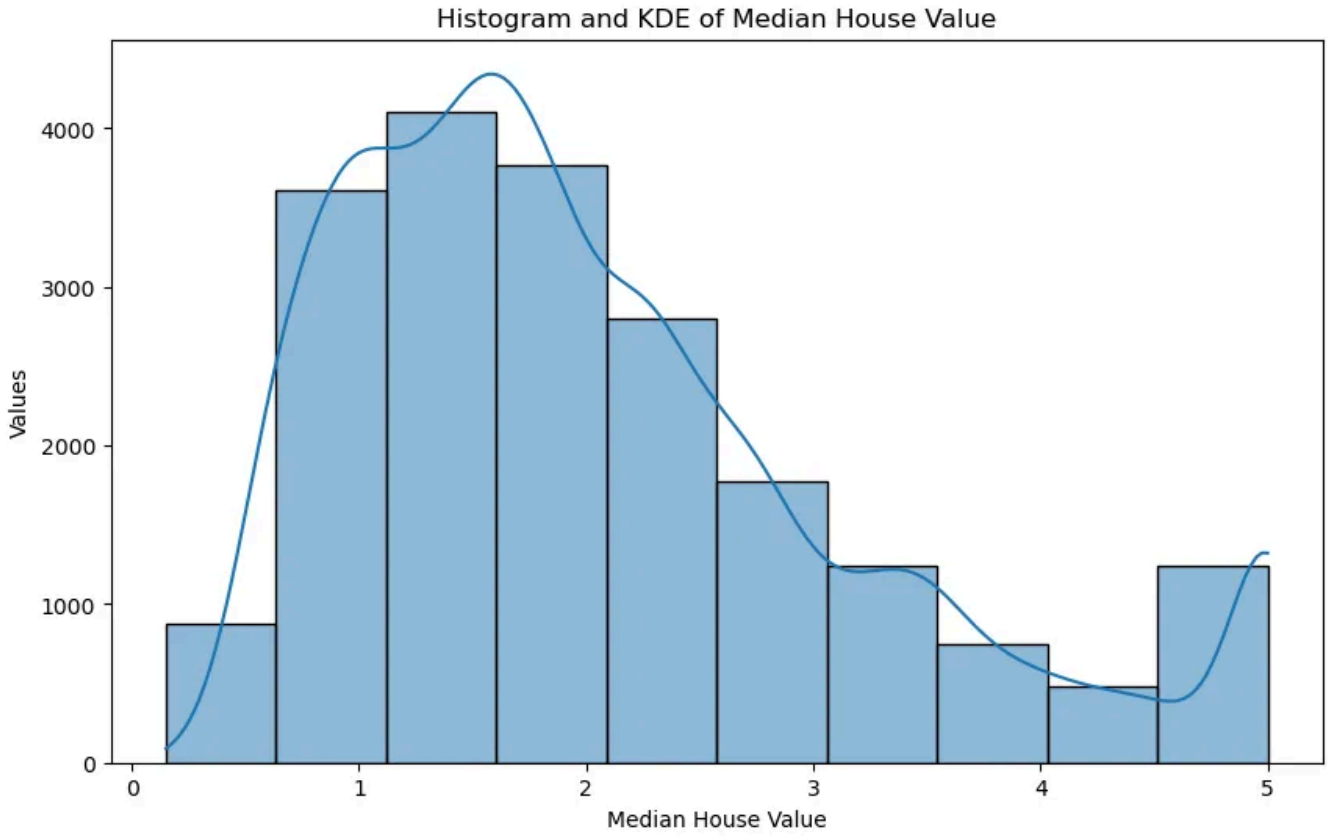
Back to top

```
# Null values for features?
print(df.isna().sum())

# What are the data types?
print(df.info())

# Histogram & KDE plot of the target variable
plt.figure(figsize=(10, 6))
sns.histplot(df['MedHouseVal'], kde=True, bins=10)
plt.title('Histogram and KDE of Median House Value')
plt.xlabel('Median House Value')
plt.ylabel('Values')
plt.show()
```

pymc1_explore_data.py hosted with ❤️ by GitHub view raw



Step 3: Prepare Data for Regression

Standardizing data can feel counterintuitive when your goal is interpretability. How will you understand the impact of a one-unit change in x on y after running your model? Fortunately, standardizing offers clear advantages in Bayesian modeling, especially with PYMC:

- **Improved Sampling Efficiency:** The No-U-Turn Sampler (NUTS) benefits from standardized data, navigating parameter space more effectively.
- **Enhanced Numerical Stability:** Standardization reduces the risk of computational issues during Markov Chain Monte Carlo (MCMC) sampling.

Addressing Interpretability

If interpretability is a priority, you can transform your coefficients back to the original scale after fitting the model. Alternatively, you can embrace the standardized coefficients and interpret the results in terms of standard deviations.

Back to top

```python
# Find Features
X = df[['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Population', 'AveOccup', 'Latitude', 'L
y = df['MedHouseVal']

# Scale features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
y_scaled = (y - y.mean()) / y.std()

# Convert scaled features to tensors
X_shared = pm.floatX(X_scaled)
y_shared = pm.floatX(y_scaled)
```

pymc1_scaling_features.py hosted with ❤ by GitHub view raw

Step 4: Build PYMC Model & Learn the Syntax

Now it's time to design our Bayesian regression model in PYMC. Below, we'll review the model's code, breaking it down line by line to understand how it works.

```python
with pm.Model() as bayesian_model:

    intercept = pm.Normal('Intercept', mu=0, sigma=1)

    beta_medinc = pm.Normal("MedianIncome", mu=1, sigma=0.5)
    beta_houseage = pm.Normal("HouseAge", mu=0.5, sigma=0.5)
    beta_averagerooms = pm.Normal("AvgRooms", mu=1, sigma=0.5)
    beta_aveoccup = pm.Normal("AvgOccupancy", mu=0, sigma=1)
    beta_latitude = pm.Normal("Latitude", mu=0, sigma=1)
    beta_longitude = pm.Normal("Longitude", mu=0, sigma=1)

    mu = (intercept
          + beta_medinc * X_shared[:, 0]
          + beta_houseage * X_shared[:, 1]
          + beta_averagerooms * X_shared[:, 2]
          + beta_aveoccup * X_shared[:, 3]
          + beta_latitude * X_shared[:, 4]
          + beta_longitude * X_shared[:, 5])

    sigma = pm.HalfNormal('sigma', sigma=1)
    y_pred = pm.Normal('y_pred', mu=mu, sigma=sigma, observed=y_shared)

    trace = pm.sample(2000, chains=4, tune=1000, target_accept=0.9)
```

pymc1_regression_model.py hosted with ❤️ by GitHub view raw

Let's dive into the details.

Line 1: Setting up the model

We start by importing PYMC using the alias `pm`. The `Model` function is used to create a model object, in our case, `bayesian_model`. Notice the use of Python's `with` statement, which serves as a context manager, allowing us to define and work with the model efficiently and cleanly. The code indented under the `with` block pertains to this specific model.

Lines 3–10: Defining priors for features

Here we define the prior distributions for the regression coefficients.

- **Why use `pm.Normal`?**
  These distributions represent our prior beliefs about the coefficients of our features, not the features themselves. For most regression problems, coefficients often follow a normal or uniform distribution.

- **Understanding `mu` and `sigma`:**
  `mu` (mean): Represents where you think the coefficient is likely centered.
  Example: Use `mu` = 0 if you think the coefficient is neutral, `mu` = 1 if positive, `mu` = -1 if negative.
  `sigma` (standard deviation): Reflects uncertainty in the coefficient. A small `sigma` suggests confidence in `mu`, while a large `sigma` allows for more variability.

- **Feature names (`beta_medinc` vs. `"MedianIncome"`):**
  The first name (e.g., `beta_medinc`) is a Python variable used in your code. The name in quotes (e.g., `"MedianIncome"`) is the label PYMC uses for plots and trace summaries. Keeping them the same can reduce confusion, but using different names can improve clarity in more complex models.

- **Feature Priors**
  Selecting meaningful priors for your features can significantly enhance model interpretability and performance. Since our features are standardized, their values already fall on a small scale, allowing us to define priors with less variance and more confidence.

- Median Income (`MedInc`): Higher median income in a district is typically associated with higher median house prices.
  mu = 1 (expect positive relationship)
  sigma = 0.5 (moderate certainty)

- House Age (`HouseAge`): Older houses often cost less than newer houses due to depreciation but it varies by neighborhood.
  mu = 0.5 (slight positive relationship)
  sigma = 0.5 (accounting for some variability)

- Average Rooms (`AveRooms`): Houses with more rooms are typically larger and therefore cost more.
  mu = 1 (positive relationship expected)
  sigma = 0.5 (more confidence in this relationship)

- Average Occupancy (`AveOccup`): This could indicate larger homes, but could also indicate lower-cost housing as well.
  mu = 0 (no strong assumption)
  sigma = 1 (high uncertainty)

- Latitude and Longitude (`Latitude, Longitude`): This feature does not inherently suggest a clear direction for their relationship with housing prices.
  mu = 0 (neutral assumption)
  sigma = 1 (high uncertainty)

- Not including Average Bedrooms (`AveBedrms`) or Population (`Population`) as they may carry less information and increase noise.

If defining each coefficient individually feels cumbersome, PYMC allows you to simplify the code by looping through features or using shared priors.

Lines 12–18: The regression formula (Likelihood)
This section defines the linear combination for the regression model ($\mu = intercept + \beta 1\, x1 + \beta 2 x2 + \ldots$). Here, `mu` represents the predicted values of our target variable. This formula doesn't have to be assigned to `mu` , but doing so makes it easier to reference in the next section.

Lines 21–22: Setting up the target variable
The target variable (`y_pred`) is modeled as a **Normal distribution** since we are predicting a continuous outcome. Key parameters include:

- `mu` : Our regression formula from the previous step.

- `sigma` : Modeled using a **Half-Normal distribution**, reflecting that standard deviations must be positive.

- `observed` : Links the model to our actual data (`y_shared`), enabling us to compare predictions against observations.

Line 23: Sampling from the posterior
This is where the magic happens. PYMC's `sample` function uses algorithms like the No-U-Turn Sampler (NUTS) to generate samples from the posterior distribution, effectively training the model. The resulting `trace` object contains everything needed to analyze and interpret the model.

- Why 2000 draws?
  This is a good default for balancing accuracy and runtime. Increasing `draws` improves precision but may require more computational resources. Don't worry

—Bayesian models don't "overfit" with additional samples, but you can monitor convergence diagnostics to ensure sampling quality.

- Why 4 chains?
  Using multiple chains is crucial for ensuring robust model convergence and sampling diversity.
  - **Improving Stability:** Multiple chains increase the likelihood of finding the true posterior by reducing the risk of getting stuck in local minima.
  - **Convergence Diagnostics:** By examining trace plots, we can verify that chains overlap and explore the posterior space effectively. Poor mixing, where the Markov Chain struggles to move through the parameter space, can be mitigated by running multiple chains.
  - **Handling Complex Models:** For models with uncertain priors, multimodal posteriors, or intricate parameter spaces, multiple chains are particularly valuable.
  - **Enhanced Inference:** Even for smaller datasets, additional chains improve sampling efficiency, reduce autocorrelation, and enhance the quality of inferences.
  In short, always use multiple chains, especially for complex models, to ensure reliable and stable results.

- Tuning Steps
  Before collecting actual samples, PYMC performs a tuning or "burn-in" phase to optimize the sampler. These initial samples are discarded to ensure the remaining samples are drawn efficiently and accurately.
  Tuning adjusts parameters like step size and scale to help the Markov Chain explore the parameter space more effectively. A well-tuned sampler leads to faster convergence and more accurate posterior estimates.

- Target acceptance rate
  The **target acceptance rate** determines how frequently the sampler accepts proposed moves through the parameter space. PYMC's default is 0.8, but increasing this rate to **0.9–0.95** can yield:
  - **Improved Stability:** Higher acceptance rates reduce the likelihood of divergent transitions.
  - **Greater Accuracy:** While slower, a higher acceptance rate ensures the model captures the posterior more precisely, especially in complex models.

For most models, increasing the acceptance rate is worth the trade-off in runtime to ensure reliable, accurate results.

Back to top

Takeaways

With the model built, we're ready to move on to diagnostics and results in the next article. Stay tuned to learn how to evaluate the model and interpret the posterior distributions!

Learn More & Connect!

GitHub
LinkedIn
PYMC Website
PYMC Developer Course for Users

Pymc | Data Science | Regression | Bayesian Statistics | Python

Follow

Written by Justin Wall

37 Followers · 98 Following

Big fan of data.

Responses (1)

Zane Falcao

What are your thoughts?

Rex
Apr 15 (edited)

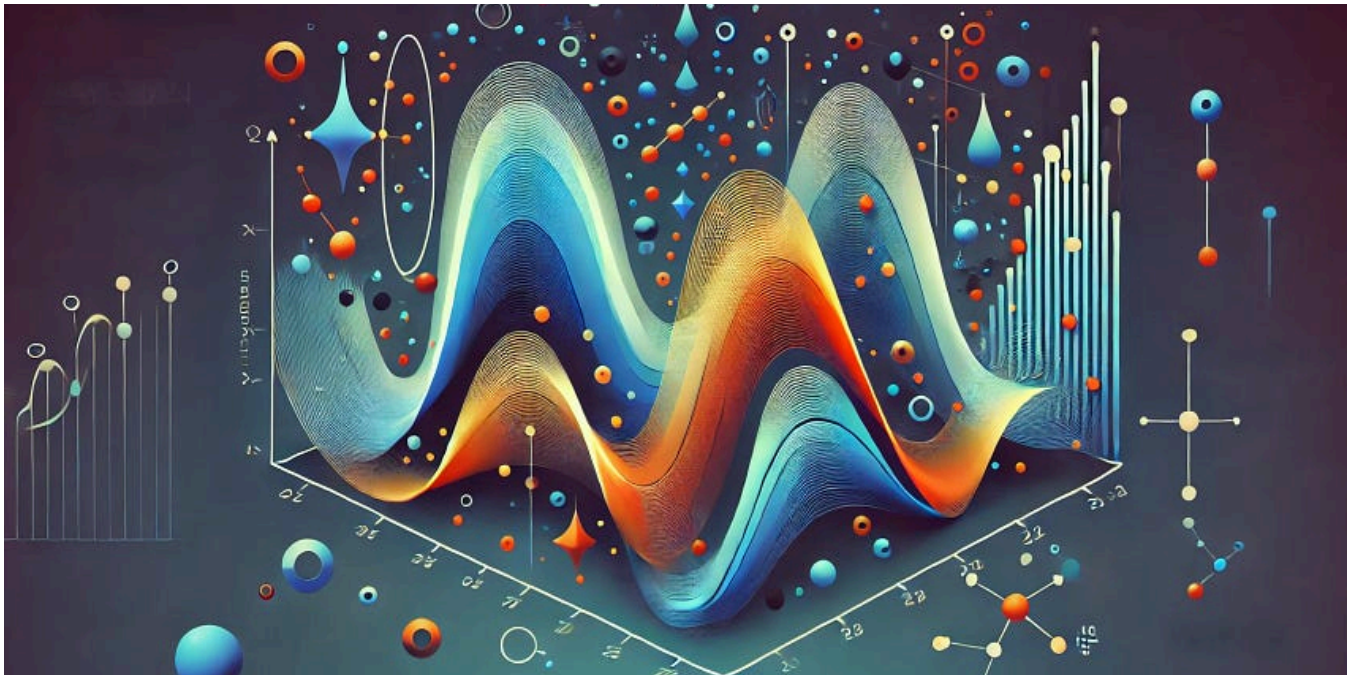
Great job explaining the setup of model in pymc (part 1). Github repository is very helpful.

I am not a member and therefore cannot see the article part 2.. However, the code are in the repository.

The code for the second part is confusing mostly... more

5 2 replies Reply

More from Justin Wall

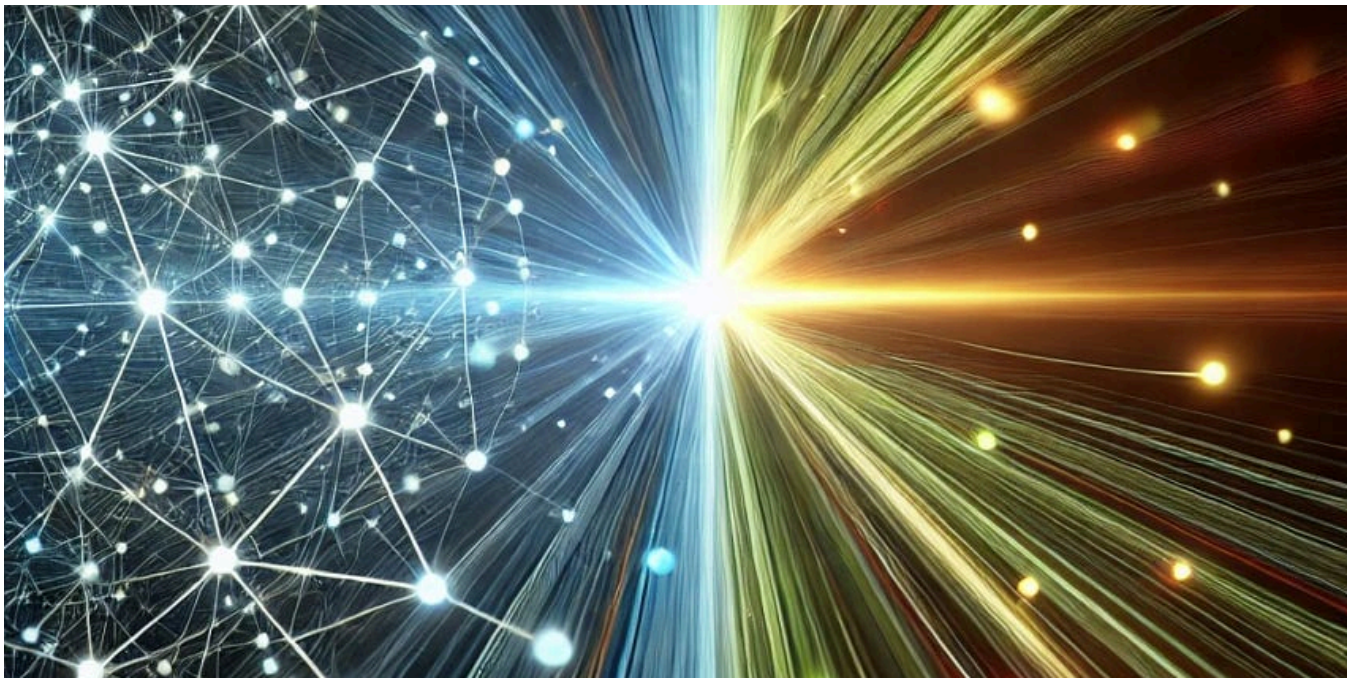


Justin Wall

Interpreting Bayesian Models: Diagnostics & Insights with PYMC

Learn how to assess your Bayesian model, interpret the posterior, and extract insights from your data.

Dec 19, 2024 32



Justin Wall

Uplift Modeling: Unlocking Causal Insights with Machine Learning

A Step-by-Step Guide to Building Incremental Response Models with an Email Marketing Dataset

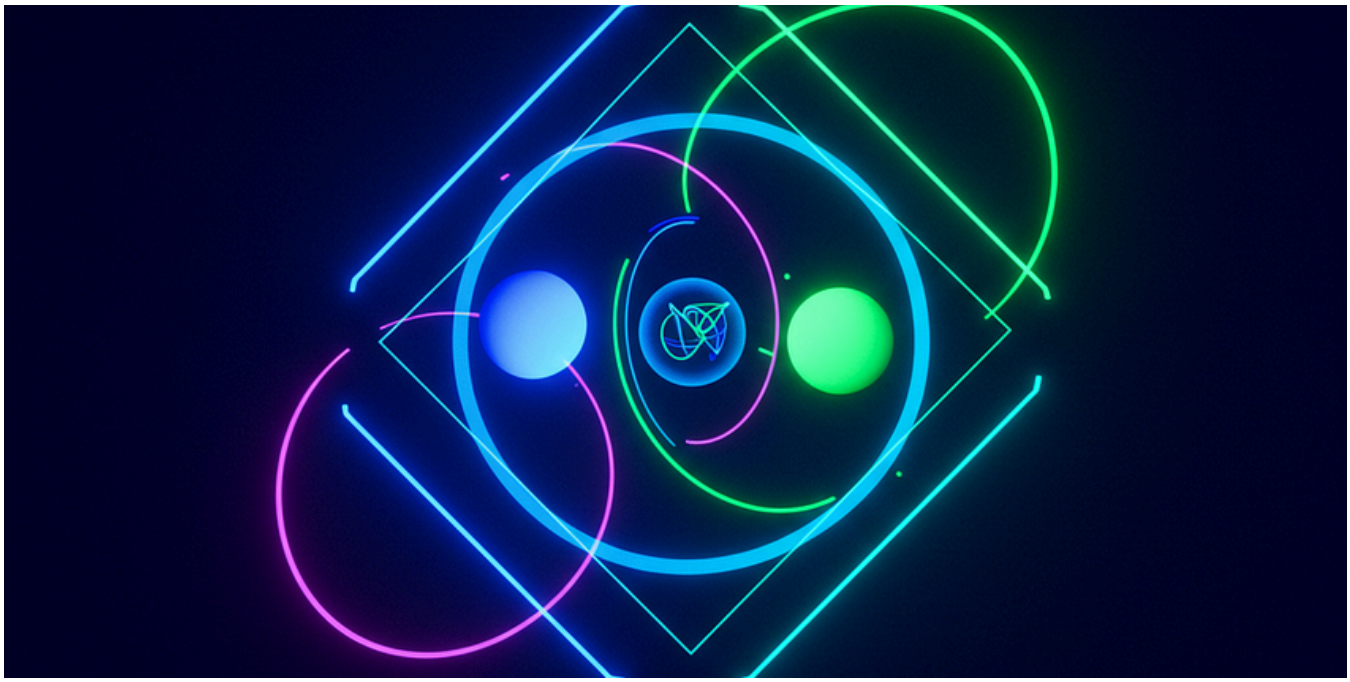
Jan 19 15

Justin Wall

Decoding the Customer Journey Using Hidden Markov Models (HMM)

Explore how Hidden Markov Models (HMMs) uncover the invisible stages behind customer actions, helping us better understand engagement...

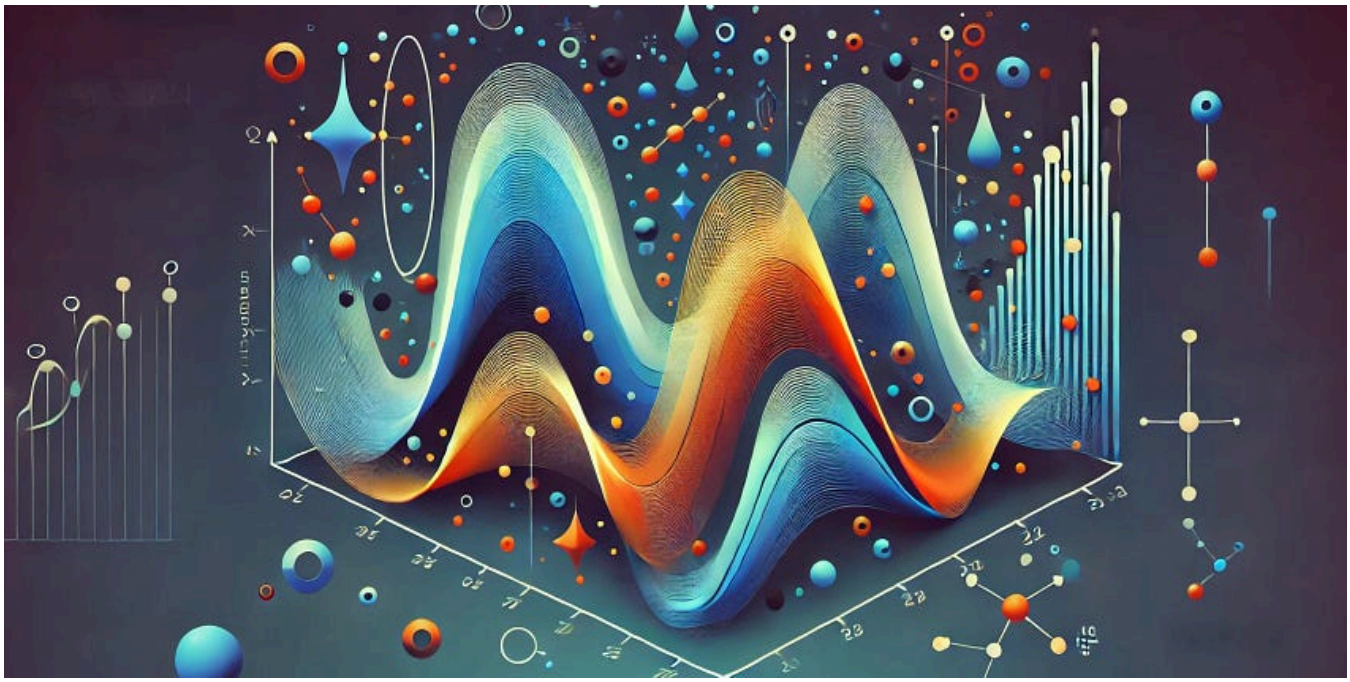
Apr 18 1



Justin Wall

The Double Delta Method: Uncovering the Real Impact of Your Loyalty Program

Recommended from Medium



Justin Wall

Interpreting Bayesian Models: Diagnostics & Insights with PYMC

Learn how to assess your Bayesian model, interpret the posterior, and extract insights from your data.

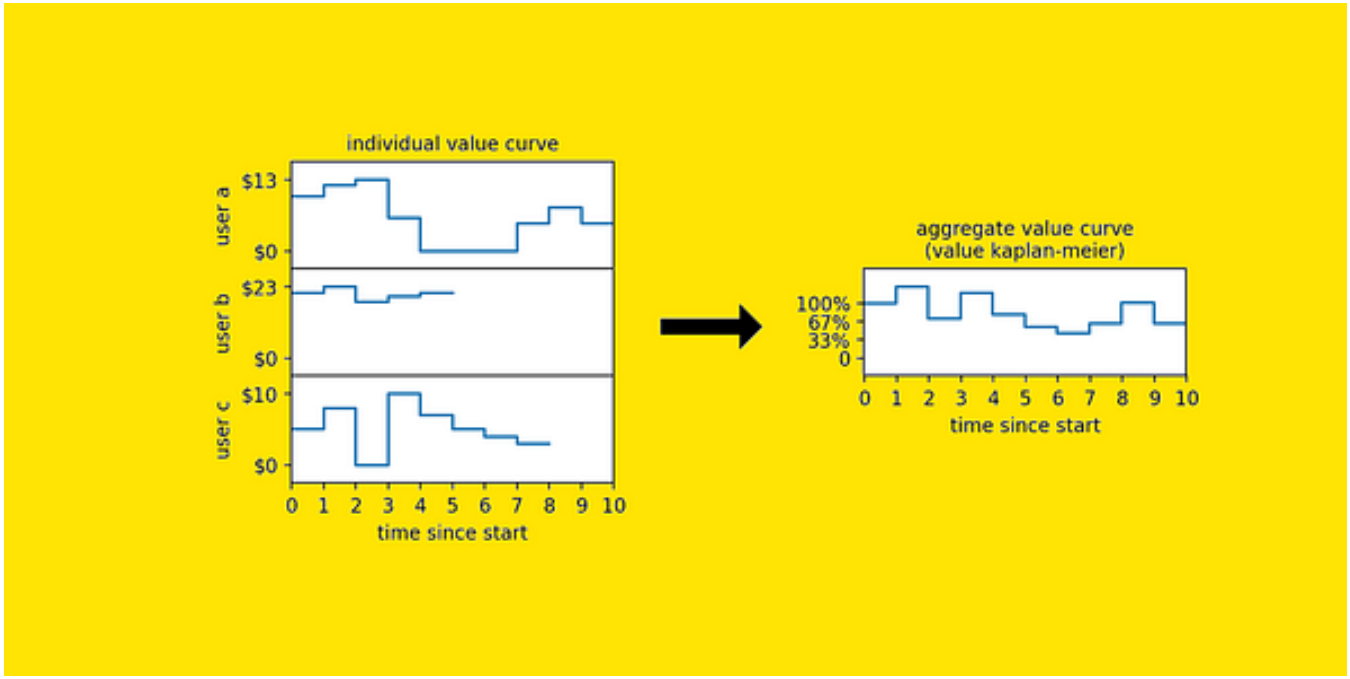
Dec 19, 2024 32

Gustavo R Santos

Price Optimization with Generalized Additive Model (GAM)

Use PyGAM to discover the price that can maximize your revenue.

Feb 5 53 2

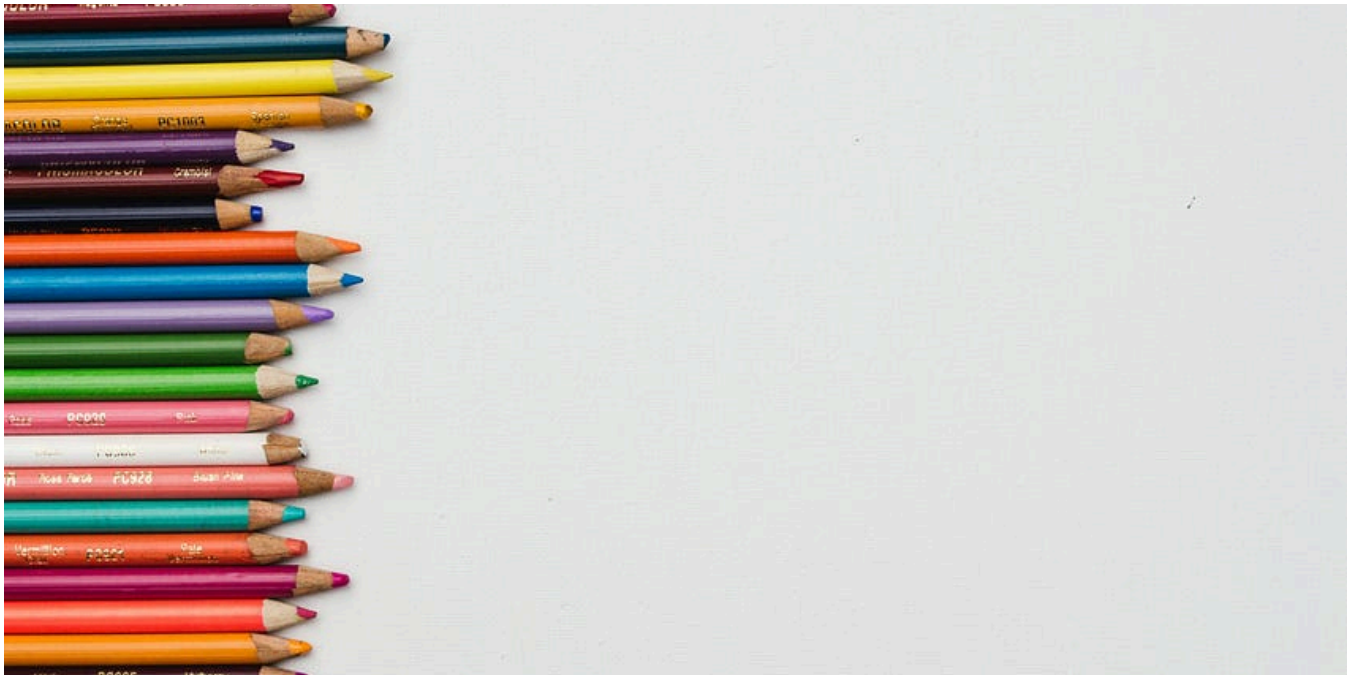


In Data Science Collective by Samuele Mazzanti

How Much Value Survives Over Time? A Value-Based Survival Curve

A generalized version of Kaplan-Meier allows to model a continuous value (like money) instead of a binary signal (like survival)

Apr 16 98 4

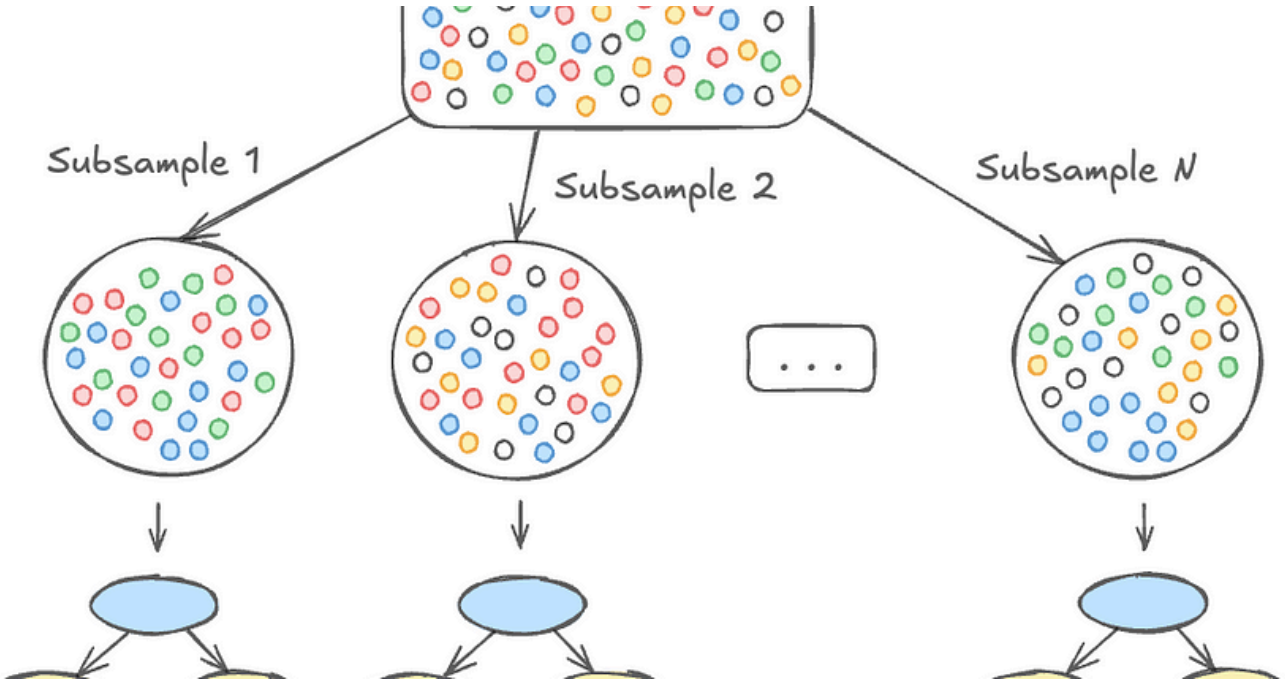


In Data And Beyond by Ben Fairbairn

Do not underestimate Linear Regression

A simple but powerful idea

Apr 16 328 9



In TDS Archive by Thomas A Dorfer

Visualizing XGBoost Parameters: A Data Scientist's Guide To Better Models

Why understanding parameters is critical to building robust models

Jan 15 686 8