Week 4
Modeling with Lines

In more than three centuries of science everything has changed except perhaps one thing: the love for the simple. – Jorge Wagensberg

Music—from classical compositions to *Sheena is a Punk Rocker* by The Ramones, passing through unrecognized hits from garage bands and Piazzolla's Libertango—is made of recurring patterns. The same scales, combinations of chords, riffs, motifs, and so on appear over and over again, giving rise to a wonderful sonic landscape capable of eliciting and modulating the entire range of emotions that humans can experience. Similarly, the universe of statistics is built upon recurring patterns, small motifs that appear now and again. In this Week, we are going to look at one of the most popular and useful of them, the **linear model** (or motif, if you want). This is a very useful model on its own and also the building block of many other models. If you've ever taken a statistics course, you may have heard of simple and multiple linear regression, logistic regression, ANOVA, ANCOVA, and so on. All these methods are variations of the same underlying motif, the linear regression model.

In this Week, we will cover the following topics:

- Simple linear regression
- NegativeBinomial regression
- Robust regression
- Logistic regression
- Variable variance
- Hierarchical linear regression
- Multiple linear regression

4.1 Simple linear regression

Many problems we find in science, engineering, and business are of the following form. We have a variable X and we want to model or predict a variable Y. Importantly, these variables are paired like $\{(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)\}$. In the most simple scenario, known as simple linear regression, both X and Y are uni-dimensional continuous random variables. By continuous, we mean a variable represented using real numbers. Using NumPy, you will represent these variables as one-dimensional arrays of floats. Usually, people call Y the dependent, predicted, or outcome variable, and X the independent, predictor, or input variable.

Some typical situations where linear regression models can be used are the following:

- Model the relationship between soil salinity and crop productivity. Then, answer questions such as: is the relationship linear? How strong is this relationship?
- Find a relationship between average chocolate consumption by country and the number of Nobel laureates in that country, and then understand why this relationship could be spurious.
- Predict the gas bill (used for heating and cooking) of your house by using the solar radiation from the local weather report. How accurate is this prediction?

In *Week 2*, we saw the Normal model, which we define as:

$$\mu \sim \text{some prior}$$

$$\sigma \sim \text{some other prior}$$

$$Y \sim \mathcal{N}(\mu, \sigma)$$

The main idea of linear regression is to extend this model by adding a predictor variable X to the estimation of the mean μ:

$$\alpha \sim \text{a prior}$$

$$\beta \sim \text{another prior}$$

$$\sigma \sim \text{some other prior}$$

$$\mu = \alpha + \beta X$$

$$Y \sim \mathcal{N}(\mu, \sigma)$$

This model says that there is a linear relation between the variable X and the variable Y. But that relationship is not deterministic, because of the noise term σ. Additionally, the model says that the mean of Y is a linear function of X, with **intercept** α and **slope** β. The intercept tells us the value of Y when X = 0 and the slope tells us the change in Y per unit change in X. Because we don't know the values of α, β, or σ we set priors distribution over them.

When setting priors for linear models we typically assume that they are independent. This assumption greatly simplifies setting priors because we then need to set three priors instead of one joint prior. At least in principle, α and β can take any value on the real line, thus it is common to use Normal priors for them. And because σ is a positive number, it is common to use a HalfNormal or Exponential prior for it.

The values the intercept can take can vary a lot from one problem to another and for different domain knowledge. For many problems I have worked on, α is usually centered around 0 and with a standard deviation no larger than 1, but this is just my experience (almost anecdotal) with a small subset of problems and not something easy to transfer to other problems. Usually, it may be easier to have an informed guess for the slope (β). For instance, we may know the sign of the slope a priori; for example, we expect the variable weight to increase, on average, with the variable height. For σ, we can set it to a large value on the scale of the variable Y , for example, two times the value for its standard deviation. We should be careful of using the observed data to guesstimate priors; usually, it is fine if the data is used to avoid using very restrictive priors. If we don't have too much knowledge of the parameter, it makes sense to ensure our prior is vague. If we instead want more informative priors, then we should not get that information from the observed data; instead, we should get it from our domain knowledge.

Extending the Normal Model

A linear regression model is an extension of the Normal model where the mean is computed as a linear function of a predictor variable.

4.2 Linear bikes

We now have a general idea of what Bayesian linear models look like. Let's try to cement that idea with an example. We are going to start very simply; we have a record of temperatures and the number of bikes rented in a city. We want to model the relationship between the temperature and the number of bikes rented. *Figure 4.1* shows a scatter plot of these two variables from the bike-sharing dataset from the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/index.php).

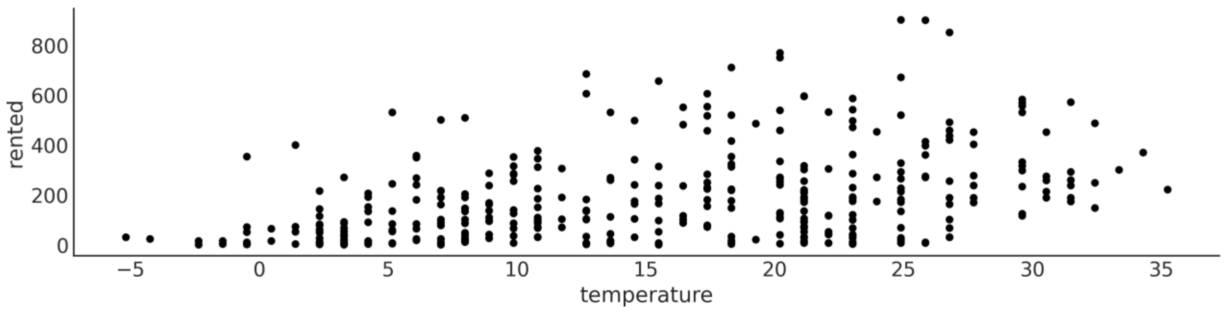

Figure 4.1: Bike-sharing dataset. Scatter plot of temperature in Celcius vs. number of rented bikes

The original dataset contains 17,379 records, and each record has 17 variables. We will only use 359 records and two variables, `temperature` (Celcius) `rented` (number of rented bikes). We are going to use`temperature` as our independent variable (our X) and the number of bikes rented as our dependent variable (our Y). We are going to use the following model:

Code 4.1

```python
with pm.Model() as model_lb:
    α = pm.Normal("α", mu=0, sigma=100)
    β  =  pm.Normal("β",  mu=0,  sigma=10)
    σ = pm.HalfCauchy("σ", 10)
    μ = pm.Deterministic("μ", α + β * bikes.temperature)
    y_pred = pm.Normal("y_pred", mu=μ, sigma=σ, observed=bikes.rented)
    idata_lb = pm.sample()
```

Take a moment to read the code line by line and be sure to understand what is going on. Also check *Figure* 4.2 for a visual representation of this model.

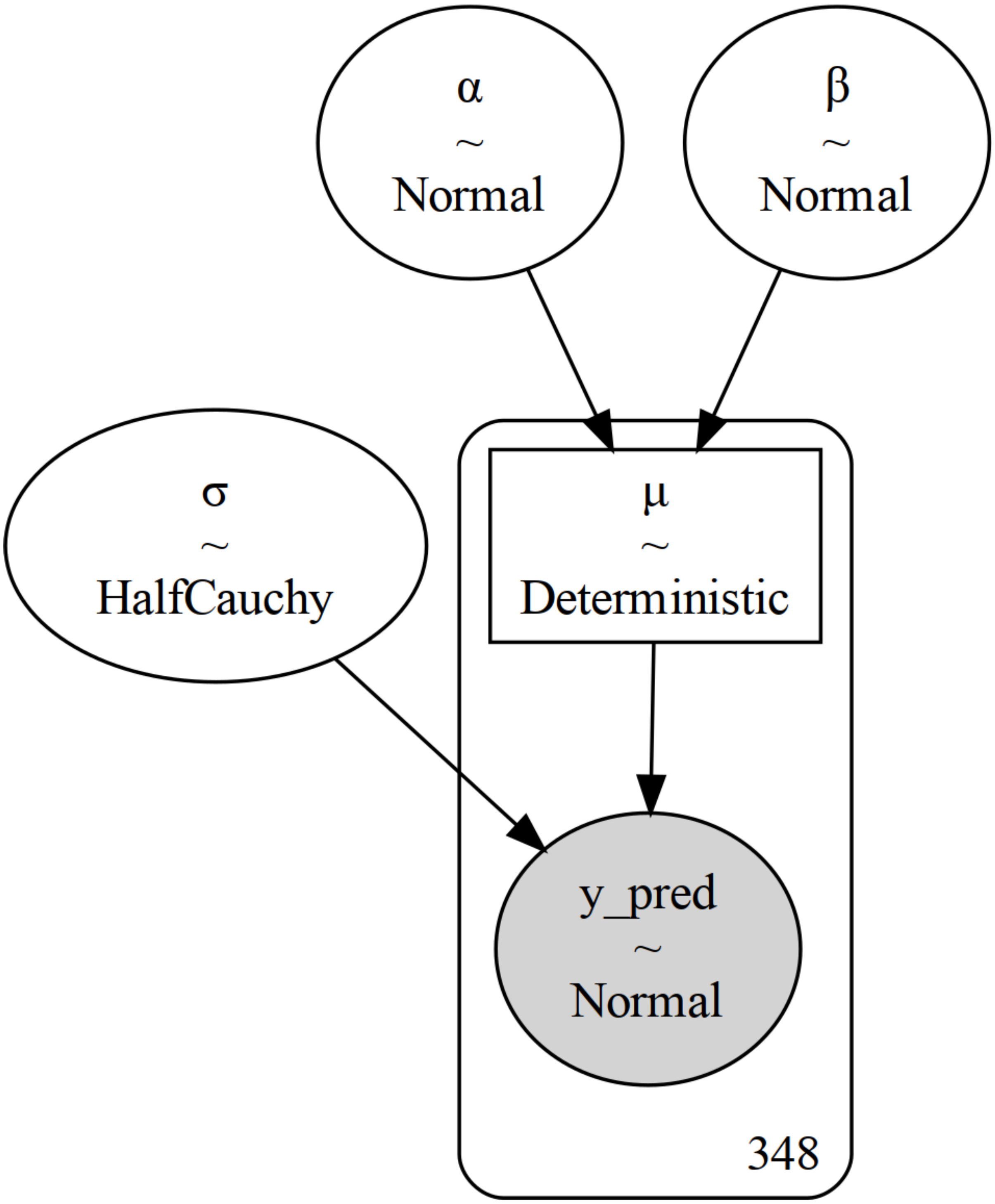

Figure 4.2: Bayesian linear model for the bike-sharing dataset

As we have previously said, this is like a Normal model, but now the mean is modeled as a linear function of the temperature. The intercept is α and the slope is β. The noise term is ϵ and the mean is μ. The only new thing here is the `Deterministic` variable μ. This variable is not a random variable, it is a deterministic variable, and it is computed from the intercept, the slope, and the temperature. We need to specify this variable because we want to save it in InferenceData for later use. We could have just written `μ = α + β * bikes.temperature` or even `_ = pm.Normal('y_pred', mu=α + β * bikes.temperature, ...` and the model will be the same, but we would not have been able to save μ in InferenceData. Notice that μ is a vector with the same length as `bikes.temperature`, which is the same as the number of records in the dataset.

4.2.1 Interpreting the posterior mean

To explore the results of our inference, we are going to generate a posterior plot but omit the deterministic variable μ. We commit it because otherwise, we would get a lot of plots, one for each value of `temperature`. We can do this by passing the names of the variables we want to include in the plot as a list to the `var_names` argument or we can negate the variable that we want to exclude as in the following block of code:

Code 4.2

```
az.plot_posterior(idata_lb, var_names=['~μ'])
```

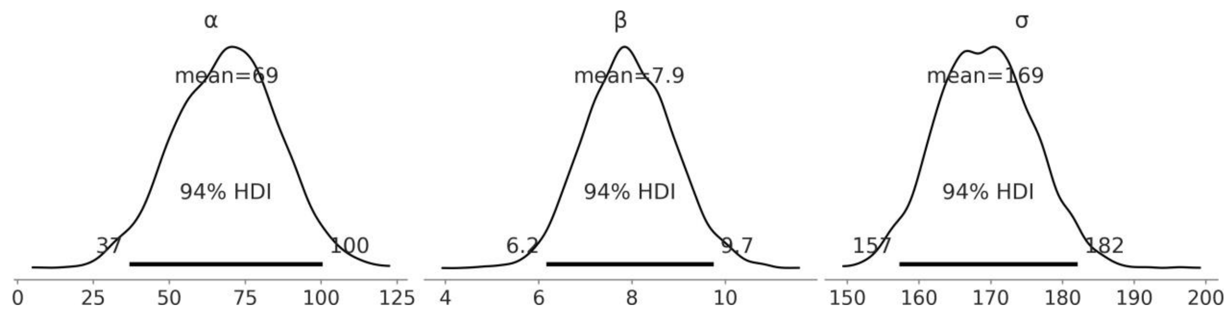

Figure 4.3: Posterior plot for the bike linear model

From *Figure 4.3*, we can see the marginal posterior distribution for α, β, and σ. If we only read the means of each distribution, say $\mu = 69 + 7.9X$, with this information we can say that the expected value of rented bikes when the temperature is 0 is 69, and for each degree of temperature the number of rented bikes increases by 7.9. So for a temperature of 28 degrees, we expect to rent $69 + 7.9 * 28 \approx 278$ bikes. This is our expectation, but the posterior also informs us about the uncertainty around this estimate. For instance, the 94% HDI for β is (6.1, 9.7), so for each degree of temperature the number of rented bikes could increase from 6 to about 10. Also even if we omit the posterior uncertainty and we only pay attention to the means, we still have uncertainty about the number of rented bikes because we have a value of σ of 170. So if we say that for a temperature of 28 degrees, we expect to rent 278 bikes, we should not be surprised if the actual number turns out to be somewhere between 100 and 500 bikes.

Now let's create a few plots that will help us visualize the combined uncertainty of these parameters. Let's start with two plots for the mean (see *Figure 4.4*). Both are plots of the mean number of rented bikes as a function of the temperature. The difference is how we represent the uncertainty. We show two popular ways of doing it. In the left subpanel, we take 50 samples from the posterior and plot them as individual lines. In the right subpanel, we instead take all the available posterior samples for μ and use them to compute the 94% HDI.

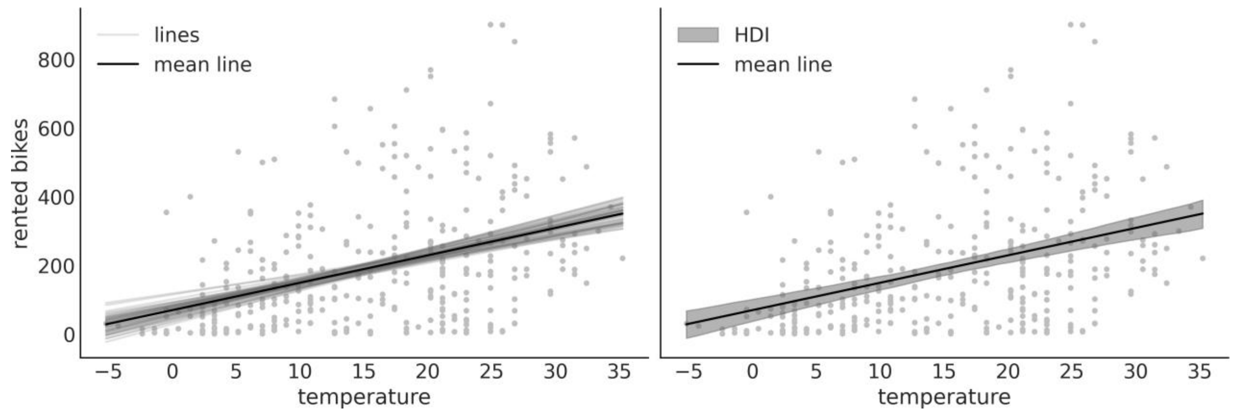

Figure 4.4: Posterior plot for the bike linear model

The plots in *Figure 4.4* convey essentially the same information, but one represents uncertainty as a set of lines and the other as a shaded area. Notice that if you repeat the code to generate the plot, you will get different lines, because we are sampling from the posterior. The shaded area, however, will be the same, because we are using all the available posterior samples. If we go further and refit the model, we will not only get different lines but the shaded area could also change, and probably the difference between runs is going to be very small; if not, you probably need to increase the number of draws, or there is something funny about your model and sampling (see *Week 10* for guidance).

Anyway, why are we showing two slightly different plots if they convey the same information? Well, to highlight that there are different ways to represent uncertainty. Which one is better? As usual, that is context-dependent. The shaded area is a good option; it is very common, and it is simple to compute and interpret. Unless there are specific reasons to show individual posterior samples, the shaded area may be your preferred choice. But we may want to show individual posterior samples. For instance, most of the lines might span a certain region, but we get a few with very high slopes. A shaded area could hide that information. When showing individual samples from the posterior it may be a good idea to animate them if you are showing them in a presentation or a video (see Kale et al. [2019] for more on this).

Another reason to show you the two plots in *Figure 4.4* is that you can learn different ways of extracting information from the posterior. Please pay attention to the next block of code. For clarity, we have omitted the code for plotting and we only show the core computations:

Code 4.3

```
posterior = az.extract(idata_lb, num_samples=50)
x_plot = xr.DataArray(
    np.linspace(bikes.temperature.min(), bikes.temperature.max(), 50),
    dims="plot_id"
)
mean_line = posterior["α"].mean() + posterior["β"].mean() * x_plot
lines = posterior["α"] + posterior["β"] * x_plot
hdi_lines = az.hdi(idata_lb.posterior["μ"])
...
```

You can see that in the first line, we used `az.extract`. This function takes the `chain` and `draw` dimensions and stacks them in a single `sample` dimension, which can be useful for later processing. Additionally, we use the `num_samples` argument to ask for a subsample from the posterior. By default, `az.extract` will operate on the posterior group. If you want to extract information from another group, you can use the `group` argument. On the second line, we define a DataArray called `x_plot`, with equally spaced values ranging from the minimum to the maximum observed temperatures. The reason to create a DataArray is to be able to use Xarray's automatic alignment capabilities in the next two lines. If we use a NumPy array, we will need to add extra dimensions, which is usually confusing. The best way to fully understand what I mean is to define `x_plot = np.linspace(bikes.temperature.min(), bikes.temperature.max())` and try to redo the plot. In the third line of code, we compute the mean of the posterior for μ for each value of `x_plot`, and in the fourth line, we compute individual values for μ. In these two lines we could have used `posterior['`μ`']`, but instead, we explicitly rewrite the linear model. We do this with the hope that it will help you to gain more intuition about linear models.

422 Interpreting the posterior predictions

What if we are not just interested in the expected (mean) value, but we want to think in terms of predictions, that is, in terms of rented bikes? Well, for that, we can do posterior predictive sampling. After executing the next line of code, `idata_lb` will be populated with a new group, `posterior_predictive`, with a variable, `y_pred`, representing the posterior predictive distribution for the number of rented bikes.

Code 4.4

```
pm.sample_posterior_predictive(idata_lb, model=model_lb, extend_inferencedata=True)
```

The black line in *Figure* 4.5 is the mean of the number of rented bikes. This is the same as in *Figure* 4.4. The new elements are the dark gray band representing the central 50% (quantiles 0.25 and 0.75) for the rented bikes and the light gray band, representing the central 94% (quantiles 0.03 and 0.97). You may notice that our model is predicting a negative number of bikes, which does not make sense. But upon reflection, this should be expected as we use a Normal distribution for the likelihood in `model_lb`. A very dirty *fix* could be to clip the predictions at values lower than 0, but that's ugly. In the next section, we will see that we can easily improve this model to avoid nonsensical predictions.

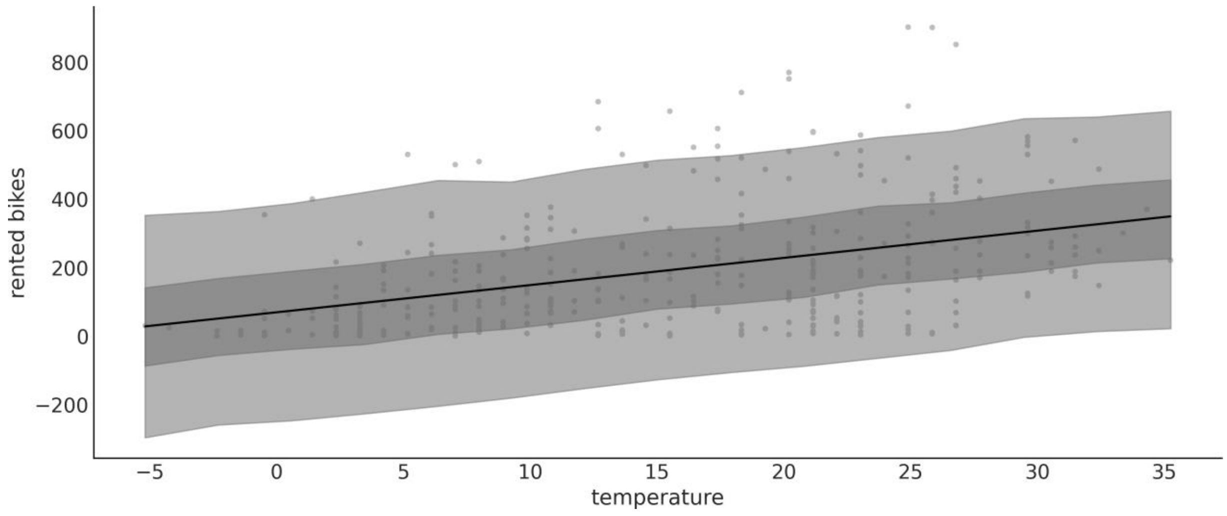

Figure 4.5: Posterior predictive plot for the bike linear model

4.3 Generalizing the linear model

The linear model we have been using is a special case of a more general model, the **Generalized Linear Model** (**GLM**). The GLM is a generalization of the linear model that allows us to use different distributions for the likelihood. At a high level, we can write a Bayesian GLM like:

$$\alpha \sim \text{a prior}$$

$$\beta \sim \text{another prior}$$

$$\theta \sim \text{some prior}$$

$$\mu = \alpha + \beta X$$

$$Y \sim \phi(f(\mu), \theta)$$

ϕ is an arbitrary distribution; some common cases are Normal, Student's t, Gamma, and NegativeBinomial. θ represents any *auxiliary* parameter the distribution may have, like σ for the Normal. We also have *f*, usually called the inverse link function. When ϕ is Normal, then *f* is the identity function. For distributions like Gamma and NegativeBinomial, *f* is usually the exponential function. Why do we need *f*? Because the linear model will generally be on the real line, but the μ parameter (or its equivalent) may be defined on a different domain. For instance, μ for the NegativeBinomial is defined for positive values, so we need to transform μ. The exponential function is a good candidate for this transformation. We are going to explore a few GLMs in this book. A good exercise for you, while reading the book, is to create a table, and every time you see a new GLM, you add one line indicating what *phi*, *theta*, and *f* are and maybe some notes about when this GLM is used. OK, let's start with our first concrete example of a GLM.

4.4 Counting bikes

How can we change `model_lb` to better accommodate the bike data? There are two things to note: the number of rented bikes is discrete and it is bounded at 0. This is usually known as count data, which is data that is the result of counting something. Count data is sometimes modeled using a continuous distribution like a Normal, especially when the number of counts is large. But it is often a good idea to use a discrete distribution. Two common choices are the Poisson and NegativeBinomial distributions. The main difference is that for Poisson, the mean and the variance are the same, but if this is not true or even approximately true, then NegativeBinomial may be a better choice as it allows the mean and variance to be different. When in doubt, you can fit both Poisson and NegativeBinomial and see which one provides a better model. We are going to do that in *Week* 5. But for now, we are going to use NegativeBinomial.

Code 4.5

```
with pm.Model() as model_neg:
    α = pm.Normal("α", mu=0, sigma=1)
    β = pm.Normal("β", mu=0, sigma=10)
    σ = pm.HalfNormal("σ", 10)
    μ = pm.Deterministic("μ", pm.math.exp(α + β * bikes.temperature))
    y_pred = pm.NegativeBinomial("y_pred", mu=μ, alpha=σ, observed=bikes.rented)
    idata_neg = pm.sample()
    idata_neg.extend(pm.sample_posterior_predictive(idata_neg))
```

The PyMC model is very similar to the previous one but with two main differences. First, we use `pm.NegativeBinomial` instead of `pm.Normal` for the likelihood. The NegativeBinomial distribution has two parameters, the mean μ and a dispersion parameter α. The variance of NegativeBinomial is $\mu + \frac{\mu^2}{\alpha}$, so the larger the value of α the larger the variance. The second difference is that μ is `pm.math.exp(α + β * bikes.temperature)` instead of just `α + β * bikes.temperature` and, as we already explained, this is needed to transform the real line into the positive interval.

The posterior predictive distribution for `model_neg` is shown in *Figure* 4.6. The posterior predictive distribution is also very similar to the one we obtained with the linear model (*Figure* 4.5). The main difference is that now we are not predicting a negative number of rented bikes! We can also see that the variance of the predictions increases with the mean. This is expected because the variance of NegativeBinomial is $\mu + \frac{\mu^2}{\alpha}$.

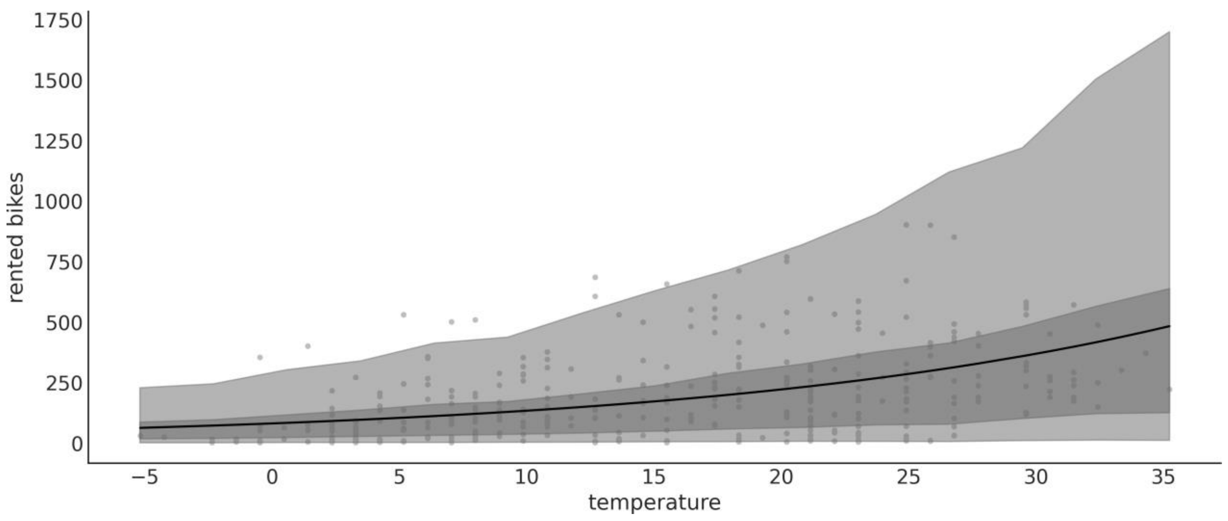

Figure 4.6: Posterior predictive plot for the bike NegativeBinomial linear model

Figure [4.7](#) shows the posterior predictive check for `model_lb` on the left and `model_neg` on the right. We can see that when using a Normal, the largest mismatch is that the model predicts a negative number of rented bikes, but even on the positive side we see that the fit is not that good. On the other hand, the NegativeBinomial model seems to be a better fit, although it's not perfect. Look at the right tail: it's heavier for the predictions than observations. But also notice that the probability of this very high demand is low. So, overall we can restate that the NegativeBinomial model is better than the Normal one.

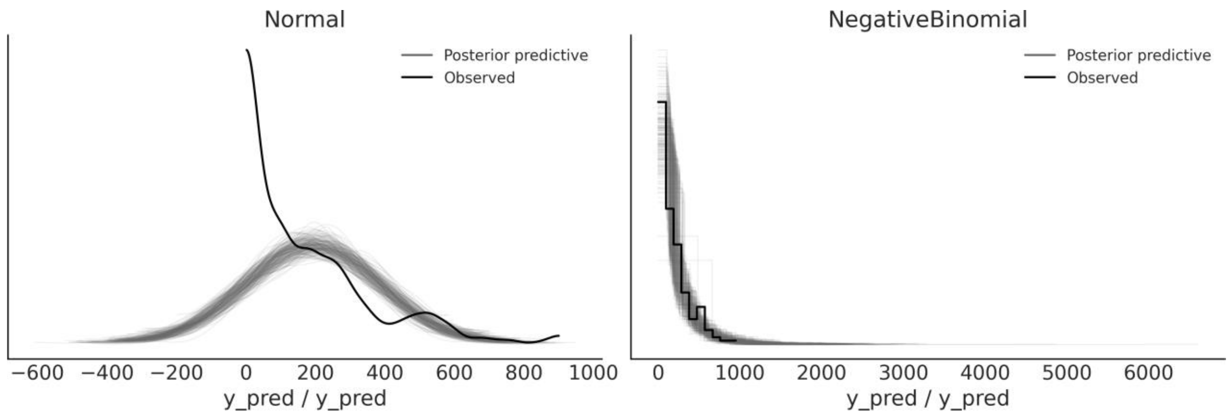

Figure 4.7: Posterior predictive check for the bike linear model

4.5 Robust regression

I once ran a complex simulation of a molecular system. At each step of the simulation, I needed it to fit a linear regression as an intermediate step. I had theoretical and empirical reasons to think that my Y was conditionally Normal given my Xs, so I decided simple linear regression should do the trick. But from time to time the simulation generated a few values of Y that were way above or below the bulk of the data. This completely ruined my simulation and I had to restart it.

Usually, these values that are very different from the bulk of the data are called outliers. The reason for the failure of my simulations was that the outliers were *pulling* the regression line away from the bulk of the data and when I passed this estimate to the next step in the simulation, the thing just halted. I solved this with the help of our good friend the Student's t-distribution, which, as we saw in *Week* 2, has heavier tails than the Normal distribution. This means that the outliers have less influence on the regression line. This is an example of a robust regression.

To exemplify the robustness that a Student's T distribution brings to linear regression, we are going to use a very simple and nice dataset: the third data group from Anscombe's quartet. If you do not know what Anscombe's quartet is, check it out on Wikipedia (https://en.wikipedia.org/wiki/Anscombe%27s_quartet).

In the following model, `model_t`, we are using a shifted exponential to avoid values close to 0. The non-shifted Exponential puts too much weight on values close to 0. In my experience, this is fine for data with none to moderate outliers, but for data with extreme outliers (or data with a few bulk points), like in Anscombe's third dataset, it is better to avoid such low values. Take this, as well as other prior recommendations, with a pinch of salt. The defaults are good starting points, but there's no need to stick to them. Other common priors are Gamma(2, 0.1) and Gamma(mu=20, sigma=15), which are somewhat similar to Exponential(1/30) but with less values closer to 0:

Code 4.6

```
with pm.Model() as model_t:
    α = pm.Normal("α", mu=ans.y.mean(), sigma=1)
    β = pm.Normal("β", mu=0, sigma=1)
    σ = pm.HalfNormal("σ", 5)
    ν_ = pm.Exponential("ν_", 1 / 29)
    ν = pm.Deterministic("ν", ν_ + 1)
    μ = pm.Deterministic("μ", α + β * ans.x)
    _ = pm.StudentT("y_pred", mu=μ, sigma=σ, nu=ν, observed=ans.y)
    idata_t = pm.sample(2000)
```

In *Figure* 4.8, we can see the robust fit, according to `model_t`, and the non-robust fit, according to SciPy's `linregress` (this function is doing least-squares regression).

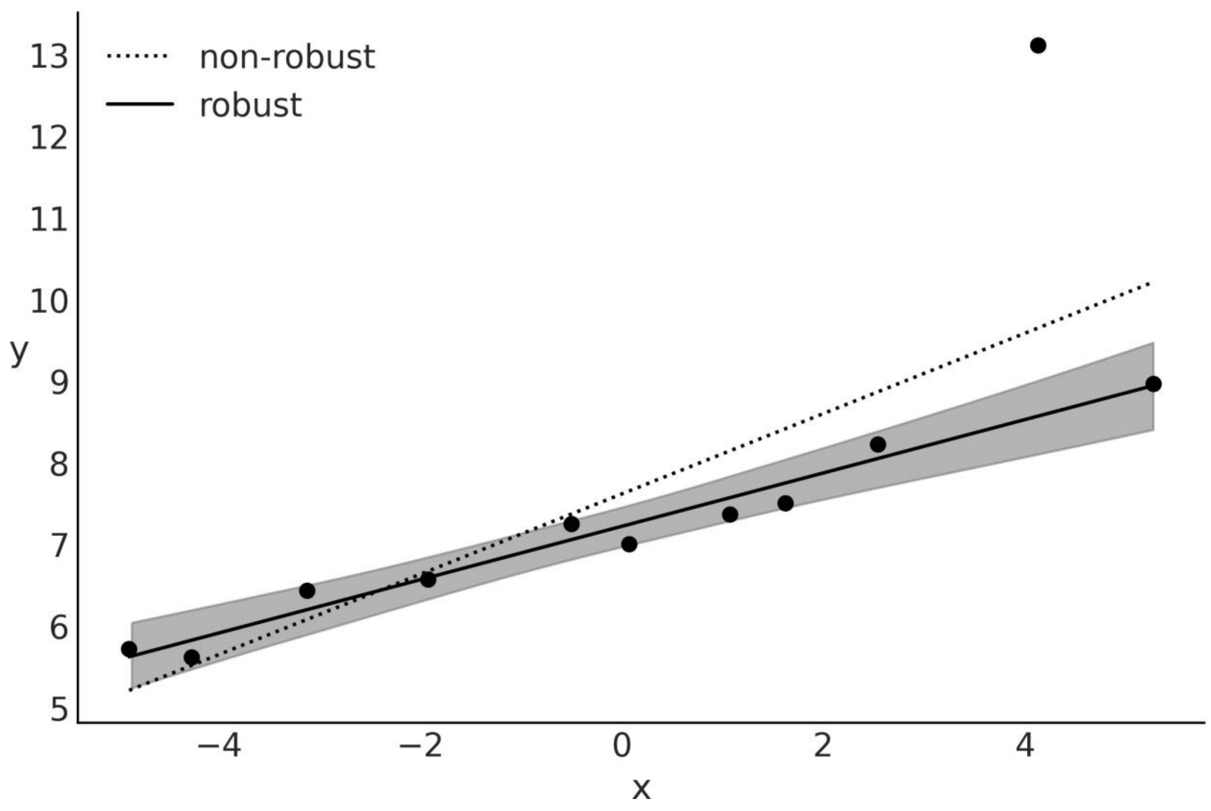

Figure 4.8: Robust regression according to `model_t`

While the non-robust fit tries to *compromise* and include all points, the robust Bayesian model, `model_t`, automatically *discards* one point and fits a line that passes closer through all the remaining points. I know this is a very peculiar dataset, but the message remains the same as for other datasets; a Student's t-distribution, due to its heavier tails, gives less importance to points that are far away from the bulk of the data.

From *Figure 4.9*, we can see that for the bulk of the data, we get a very good match. Also, notice that our model predicts values away from the bulk to both sides and not just above the bulk (as in the observed data). For our current purposes, this model is performing just fine and it does not need further changes. Nevertheless, notice that for some problems, we may want to avoid this. In such a case, we should probably go back and change the model to restrict the possible values of `y_pred` to positive values using a truncated Student's t-distribution. This is left as an exercise for the reader.

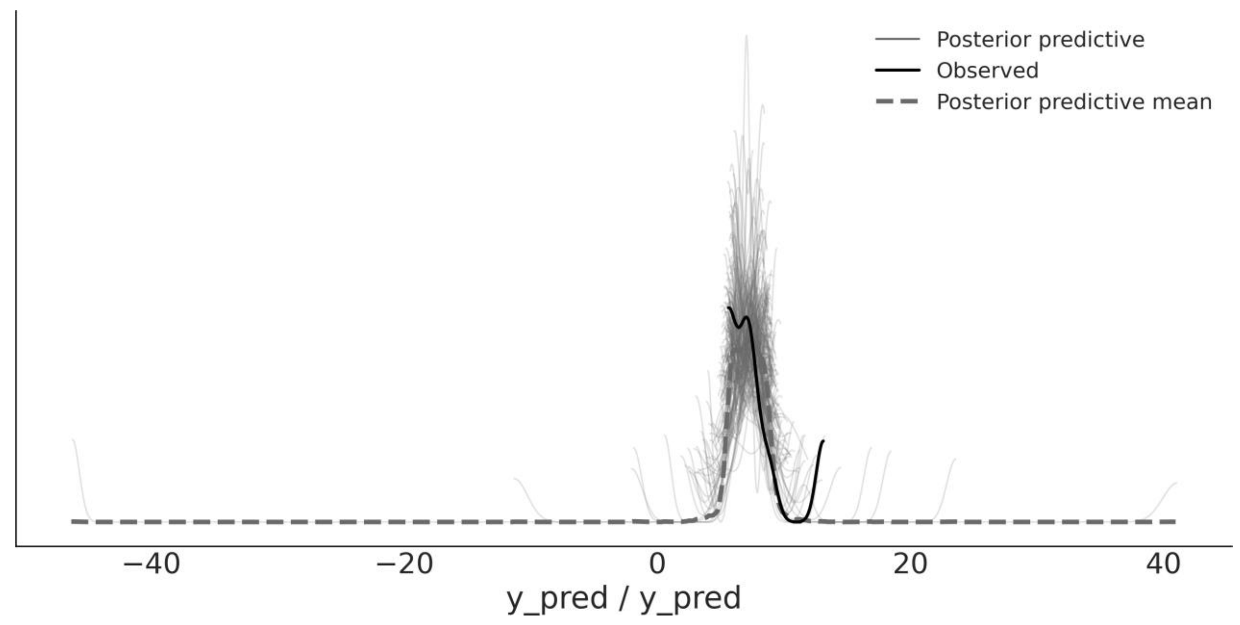

Figure 4.9: Posterior predictive check for `model_t`

4.6 Logistic regression

The logistic regression model is a generalization of the linear regression model, which we can use when the response variable is binary. This model uses the logistic function as an inverse link function. Let's get familiar with this function before we move on to the model:

$$\text{logistic}(z) = \frac{1}{1 + e^{-z}}$$

For our purpose, the key property of the logistic function is that irrespective of the values of its argument *z*, the result will always be a number in the [0-1] interval. Thus, we can see this function as a convenient way to compress the values computed from a linear model into values that we can feed into a Bernoulli distribution. This logistic function is also known as the sigmoid function because of its characteristic S-shaped aspect, as we can see from *Figure 4.10*.

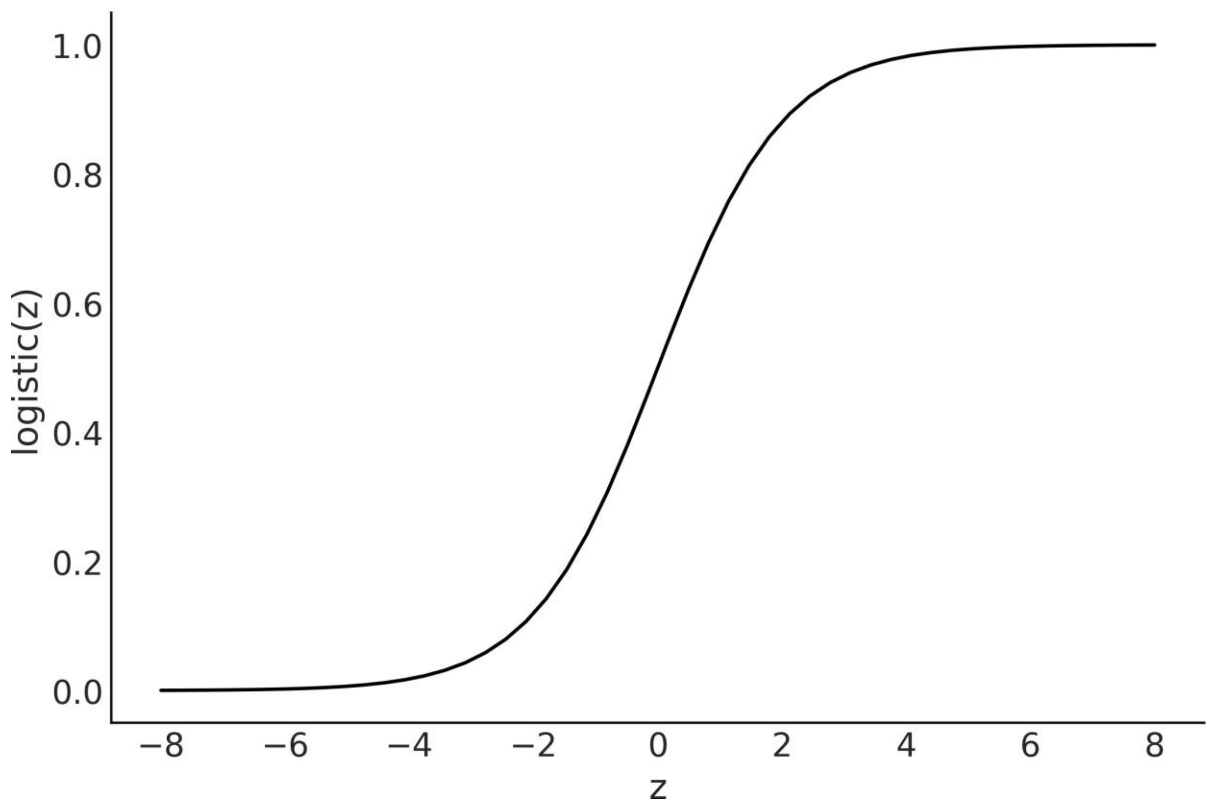

Figure 4.10: Logistic function

4.6.1 The logistic model

We have almost all the elements to turn a simple linear regression into a simple logistic regression. Let's begin with the case of only two classes, for example, ham/spam, safe/unsafe, cloudy/sunny, healthy/ill, or hotdog/not hotdog. First, we codify these classes by saying that the predicted variable y can only take two values, 0 or 1, that is $y \in \{0,1\}$.

Stated this way, the problem sounds very similar to the coin-flipping one we used in previous Weeks. We may remember we used the Bernoulli distribution as the likelihood. The difference with the coin-flipping problem is that now θ is not going to be generated from a beta distribution; instead, θ is going to be defined by a linear model with the logistic as the inverse link function. Omitting the priors, we have:

$$\theta = \text{logistic}(\alpha + \beta x)$$

$$y \sim \text{Bernoulli}(\theta)$$

We are going to apply logistic regression to the classic iris dataset which has measurements from flowers from three closely related species: setosa, virginica, and versicolor. These measurements are the petal length, petal width, sepal length, and sepal width. In case you are wondering, sepals are modified leaves whose function is generally related to protecting the flowers in a bud.

We are going to begin with a simple case. Let's assume we only have two classes, setosa, and versicolor, and just one independent variable or feature, `sepal_length`. We want to predict the probability of a flower being setosa given its sepal length.

As is usually done, we are going to encode the `setosa` and `versicolor` categories with the numbers `0` and `1`. Using pandas, we can do the following:

Code 4.7

```
df = iris.query("species == ('setosa', 'versicolor')")
y_0 = pd.Categorical(df["species"]).codes
x_n = "sepal_length"
x_0 = df[x_n].values
x_c = x_0 - x_0.mean()
```

As with other linear models, centering the data can help with the sampling. Now that we have the data in the right format, we can finally build the model with PyMC:

Code 4.8

```
with pm.Model() as model_lrs:
    α = pm.Normal("α", mu=0, sigma=1)
    β = pm.Normal("β", mu=0, sigma=5)
    μ = α + x_c * β
    θ = pm.Deterministic("θ", pm.math.sigmoid(μ))
    bd = pm.Deterministic("bd", -α / β)
    yl = pm.Bernoulli("yl", p=θ, observed=y_0)

    idata_lrs = pm.sample()
```

`model_lrs` has two deterministic variables: θ and `bd`. θ is the result of applying the logistic function to variable μ. `bd` is the boundary decision, which is the value we use to separate classes. We will discuss this later in detail. Another point worth mentioning is that instead of writing the logistic function ourselves, we are using the one provided by PyMC, `pm.math.sigmoid`.

Figure 4.11 shows the result of `model_lrs`:

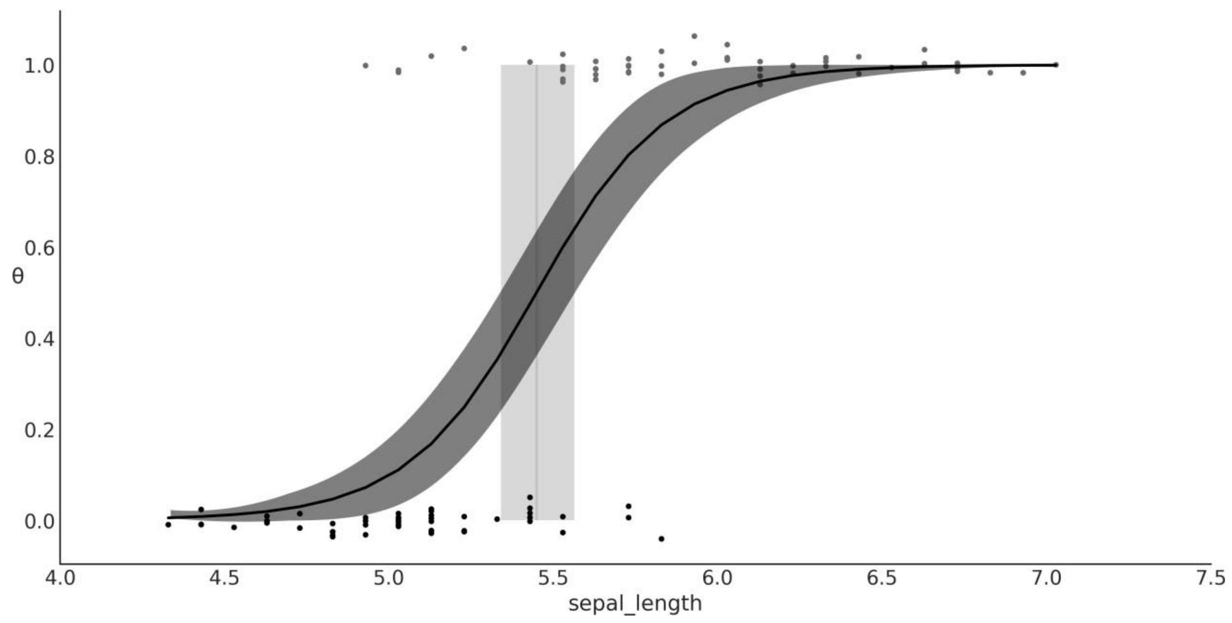

Figure 4.11: Logistic regression, result of `model_lrs`

Figure 4.11 shows the sepal length versus the probability of being versicolor θ (and if you want, also the probability of being setosa, $1-\theta$). We have added some jitter (noise) to the binary response so the point does not overlap. An S-shaped (black) line is the mean value of θ. This line can be interpreted as the probability of a flower being versicolor, given that we know the value of the sepal length. The semitransparent S-shaped band is the 94% HDI. What about the vertical line? That's the topic of the next section.

462 Classification with logistic regression

Logistic regression, a model that, despite its name, is generally framed as a method for solving classification problems. Let's see the source of this duality.

Regression problems are about predicting a continuous value for an output variable given the values of one or more input variables. We have seen many examples of regression that include logistic regression. However, logistic regression is usually discus

sed in terms of classification. Classification involves assigning discrete values (representing a class, like versicolor) to an output variable given some input variables, for instance, stating that a flower is versicolor or setosa given its sepal length.

So, is logistic regression a regression or a classification method? The answer is that it is a regression method; we are regressing the probability of belonging to some class, but it can be used for classification too. The only thing we need is a decision rule: for example, we assign the class `versicolor` if $\theta \geq 0.5$

and assign `setosa` otherwise. The vertical line in *Figure* 4.11 is the boundary decision, and it is defined as the value of the independent variable that makes the probability of being versicolor equal to 0.5. We can calculate this value analytically, and it is equal to $-\frac{\alpha}{\beta}$. This calculation is based on the definition of the model:

$$\theta = \text{logistic}(\alpha + \beta x)$$

And from the definition of the logistic function, we have that $\theta = 0.5$ when $\alpha + \beta x = 0$.

$$0.5 = \text{logistic}(\alpha + \beta x) \iff 0 = \alpha + \beta x$$

Reordering, we find that the value of x that makes $\theta = 0.5$ is $-\frac{\alpha}{\beta}$.

Because we have uncertainty in the value of α and β, we also have uncertainty about the value of the boundary decision. This uncertainty is represented as the vertical (gray) band in *Figure* 4.11, which goes from ≈ 5.3 to ≈ 5.6. If we were doing automatic classification of flowers based on their sepal length (or any similar problem that could be framed within this model), we could assign setosa to flowers with a sepal length below 5.3 and versicolor to flowers with sepal length above 5.6. For flowers with a sepal lengths between 5.3 and 5.6, we would be uncertain about their class, so we could either assign them randomly or use some other information to make a decision, including asking a human to check the flower.

To summarize this section:

- The value of θ is, generally speaking, $P(Y = 1|X)$. In this sense, logistic regression is a true regression; the key detail is that we are regressing the probability that a data point belongs to class 1, given a linear combination of features.
- We are modeling the mean of a dichotomous variable, which is a number in the [0-1] interval. Thus, if we want to use logistic regression for classification, we need to introduce a rule to turn this probability into a two-class assignment. For example, if $P(Y = 1) > 0.5$, we assign that observation to class 1, otherwise we assign it to class 0.
- There is nothing special about the value of 0.5, other than that it is the number in the middle of 0 and 1. This boundary can be justified when we are OK with misclassifying a data point in either direction. But this is not always the case, because the cost associated with the misclassification does not need to be symmetrical. For example, if we are trying to predict whether a patient has a disease or not, we may want to use a boundary that minimizes the number of false negatives (patients that have the disease but we predict they don't) or false positives (patients that don't have the disease but we predict they do). We will discuss this in more detail in the next section.

4.6.3 Interpreting the coefficients of logistic regression

We must be careful when interpreting the coefficients of logistic regression. Interpretation is not as straightforward as with simple linear models. Using the logistic inverse link function introduces a non-linearity that we have to take into account. If β is positive, increasing x will increase $p(y = 1)$ by some amount, but the amount is not a linear function of x. Instead, the dependency is non-linear on the value of x, meaning that the effect of x on $p(y = 1)$ depends on the value of x. We can visualize this fact in *Figure* 4.11. Instead of a line with a constant slope, we have an S-shaped line with a slope that changes as a function of x.

A little bit of algebra can give us some further insight into how much $p(y = 1)$ changes with x. The basic logistic model is:

$$\theta = \text{logistic}(\alpha + \beta x)$$

The inverse of the logistic is the logit function, which is:

$$\text{logit}(z) = \log \frac{z}{1 - z}$$

Combining these two expressions, we get:

$$\text{logit}(\theta) = \log \frac{\theta}{1 - \theta} = \alpha + \beta x$$

Remember that θ in our model is $p(y = 1)$, so we can rewrite the previous expression as:

$$\log \left(\frac{p(y = 1)}{1 - p(y = 1)} \right) = \alpha + \beta x$$

The $\frac{p(y=1)}{1-p(y=1)}$ quantity is known as the **odds** of $y = 1$. If we call $y = 1$ a *success*, then the odds of success is the ratio of the probability of success over the probability of failure. For example, while the probability of getting a 2 by rolling a fair die is $\frac{1}{6}$, the odds of getting a 2 are $\frac{1/6}{5/6} = \frac{1}{5} = 0.2$. In other words, there is one favorable event for every five unfavorable events. Odds are often used by gamblers because they provide a more intuitive tool to think about bets than raw probabilities. *Figure* 4.12 shows how probabilities are related to odds and log-odds.

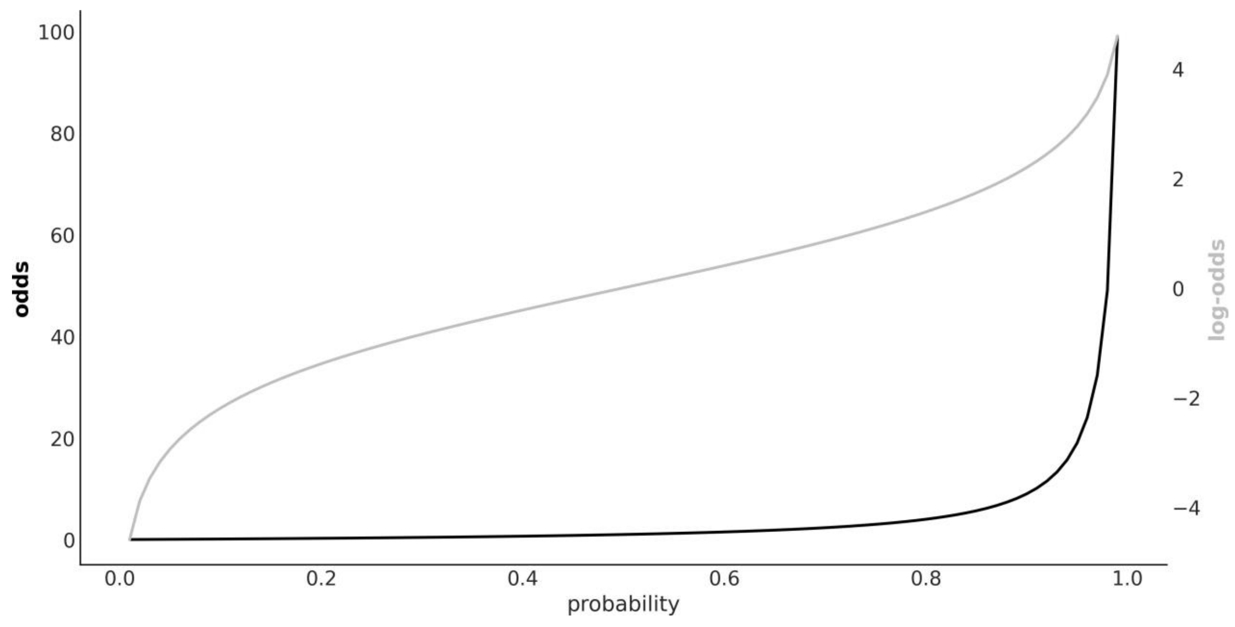

Figure 4.12: Relationship betw een probability, odds, and log-odds

Interpreting Logistic Regression

In logistic regression, the β coefficient (the *slope*) encodes the increase in log-odds units by a unit increase of the x variable.

The transformation from probability to odds is a monotonic transformation, meaning the odds increase as the probability increases, and the other way around. While probabilities are restricted to the [0,1] interval, odds live in the [0,∞) interval. The logarithm is another monotonic transformation and log-odds are in the (−∞,∞) interval.

4.7 Variable variance

We have been using the linear motif to model the mean of a distribution and, in the previous section, we used it to model interactions. In statistics, it is said that a linear regression model presents heteroskedasticity when the variance of the errors is not constant in all the observations made. For those cases, we may want to consider the variance (or standard deviation) as a (linear) function of the dependent variable.

The World Health Organization and other health institutions around the world collect data for newborns and toddlers and design growth chart standards. These charts are an essential component of the pediatric toolkit and also a measure of the general well-being of populations to formulate health-related policies, plan interventions, and monitor their effectiveness. An example of such data is the lengths (heights) of newborn/toddler girls as a function of their age (in months):

Code 4.9

```
data = pd.read_csv("data/babies.csv")
data.plot.scatter("month",  "length")
```

To model this data, we are going to introduce three elements we have not seen before:

- σ is now a linear function of the predictor variable. Thus, we add two new parameters, γ and δ. These are direct analogs of α and β in the linear model for the mean.
- The linear model for the mean is a function of $\sqrt(X)$. This is just a simple trick to fit a linear model to a curve.
- We define a `MutableData` variable, `x_shared`. Why we want to do this will become clear soon.

Our full model is:

Code 4.10

```
with pm.Model() as model_vv:
    x_shared = pm.MutableData("x_shared", data.month.values.astype(float))
    α = pm.Normal("α", sigma=10)
    β = pm.Normal("β", sigma=10)
    γ = pm.HalfNormal("γ", sigma=10)
    δ = pm.HalfNormal("δ", sigma=10)

    μ = pm.Deterministic("μ", α + β * x_shared**0.5)
    σ = pm.Deterministic("σ", γ + δ * x_shared)

    y_pred = pm.Normal("y_pred", mu=μ, sigma=σ, observed=data.length)

    idata_vv = pm.sample()
```

On the left panel of *Figure 4.13*, we can see the mean of μ represented by a black curve, and the two semi-transparent gray bands represent one and two standard deviations. On the right panel, we have the estimated variance as a function of the length. As you can see, the variance increases with the length, which is what we expected.

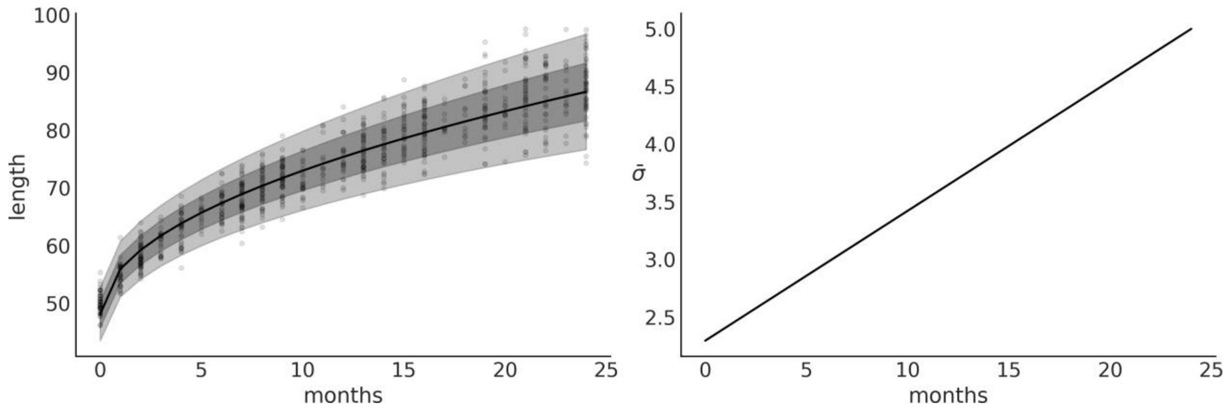

Figure 4.13: Posterior fit for `model_vv` on the left panel. On the right is the mean estimated variance as a function of the length

Now that we have fitted the model, we might want to use the model to find out how the length of a particular girl compares to the distribution. One way to answer this question is to ask the model for the distribution of the variable `length` for babies of, say, 0.5 months. We can answer this question by sampling from the posterior predictive distribution conditional on a length of 0.5. Using PyMC, we can get the answer by sampling `pm.sample_posterior_predictive`; the only problem is that by default, this function will return values of $\tilde{y}$ for the already observed values of x, i.e., the values used to fit the model. The easiest way to get predictions for unobserved values is to define a `MutableData` variable (`x_shared` in the example) and then update the value of this variable right before sampling the posterior predictive distribution, as shown in the following code block:

Code 4.11

```python
with model_vv:
    pm.set_data({"x_shared": [0.5]})
    ppc = pm.sample_posterior_predictive(idata_vv)
    y_ppc = ppc.posterior_predictive["y_pred"].stack(sample=("chain", "draw"))
```

Now we can plot the expected distribution of lengths for 2-week-old girls and calculate other quantities, like the percentile for a girl of that length (see *Figure 4.14*).

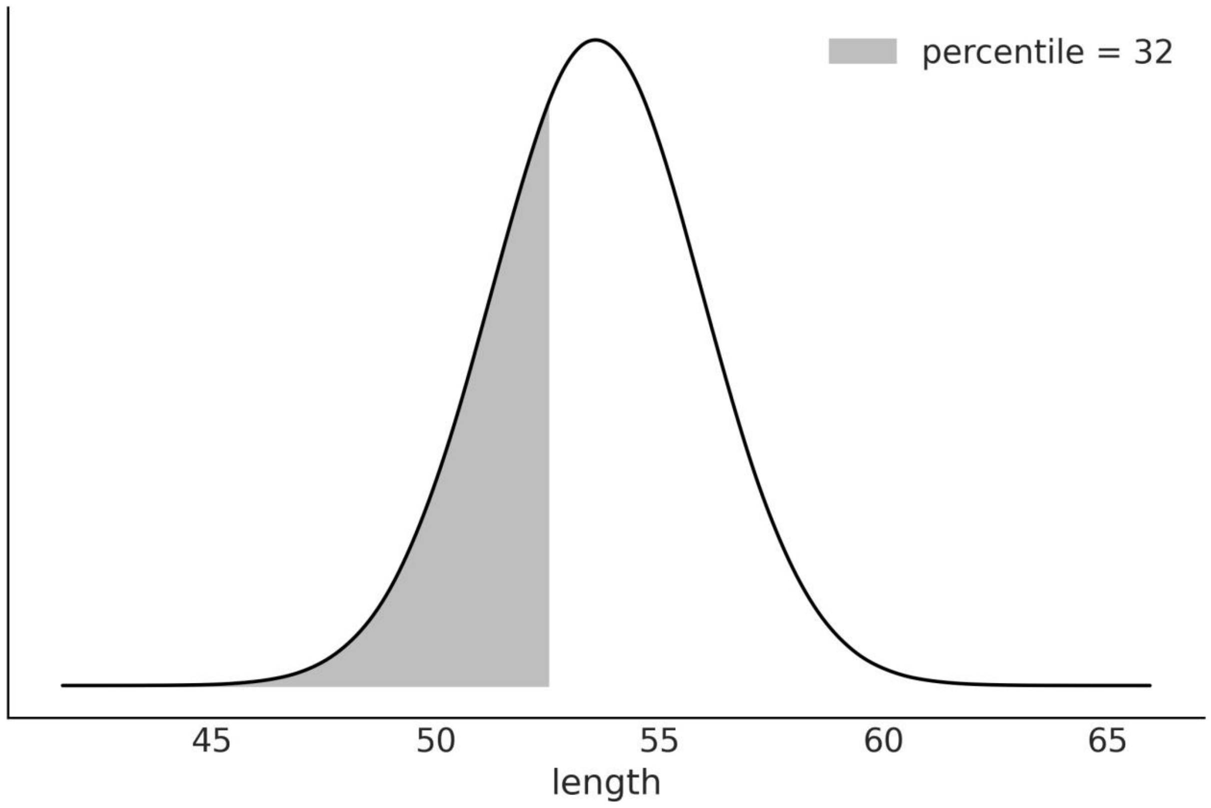

Figure 4.14: Expected distribution of length at 0.5 months. The shaded area represents 32% of the accumulated mass

4.8 Hierarchical linear regression

In *Week 3*, we learned the rudiments of hierarchical models, a very powerful concept that allows us to model complex data structures. Hierarchical models allow us to deal with inferences at the group level and estimations above the group level. As we have already seen, this is done by including hyperpriors. We also showed that groups can share information by using a common hyperprior and this provides shrinkage, which can help us to regularize the estimates.

We can apply these very same concepts to linear regression to obtain hierarchical linear regression models. In this section, we are going to walk through two examples to elucidate the application of these concepts in practical scenarios. The first one uses a synthetic dataset, and the second one uses the `pigs` dataset.

For the first example, I have created eight related groups, including one group with just one data point. We can see what the data looks like from *Figure 4.15*.

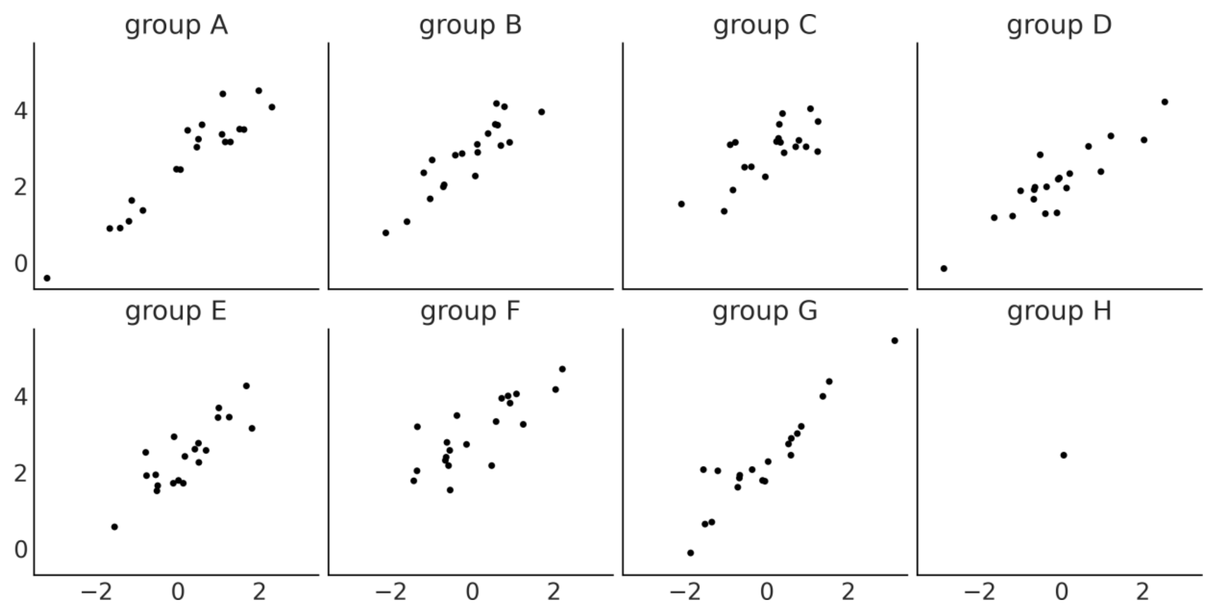

Figure 4.15: Synthetic data for the hierarchical linear regression example

First, we are going to fit a non-hierarchical model:

Code 4.12

```
coords = {"group": ["A", "B", "C", "D", "E", "F", "G", "H"]}

with pm.Model(coords=coords) as unpooled_model:
    α = pm.Normal("α", mu=0, sigma=10, dims="group")
    β = pm.Normal("β", mu=0, sigma=10, dims="group")
    σ = pm.HalfNormal("σ", 5)
    _ = pm.Normal("y_pred", mu=α[idx] + β[idx] * x_m, sigma=σ, observed=y_m)

    idata_up = pm.sample()
```

Figure 4.16 shows the posterior estimated values for the parameters α and β.

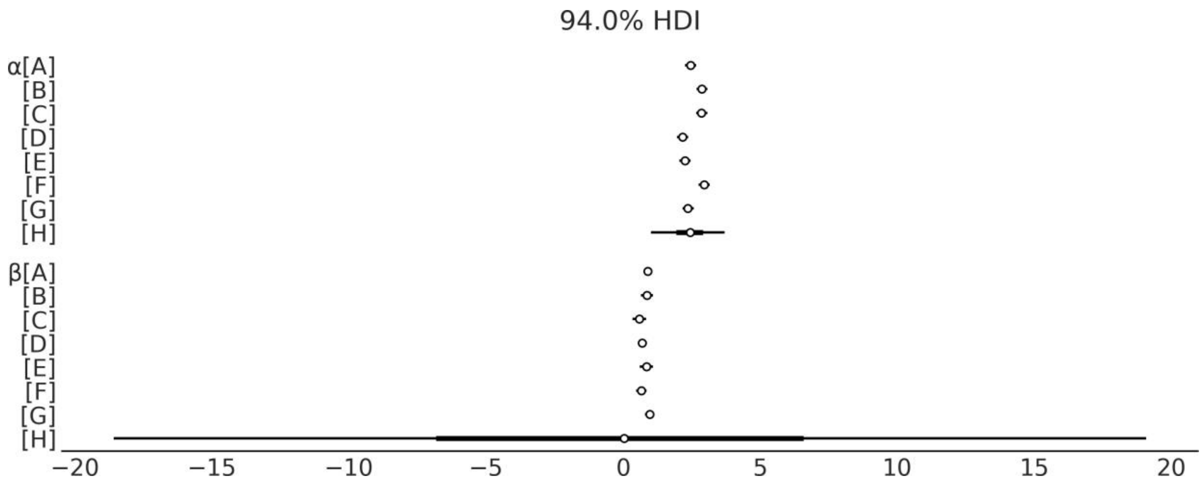

Figure 4.16: Posterior distribution for *α* and *β* for `unpooled_model`

As you can see from *Figure 4.16* the estimates for group `H` are very different from the ones for the other groups. This is expected as for group `H`, we only have one data point, that is we do not have enough information to fit a line. We need at least two points; otherwise, the model will be over-parametrized, meaning we have more parameters than the ones we can determine from the data alone.

To overcome this situation we can provide some more information; we can do this by using priors or by adding more structure to the model. Let's add more structure by building a hierarchical model.

This is the PyMC model for the hierarchical model:

Code 4.13

```
with pm.Model(coords=coords) as hierarchical_centered:
    # hyperpriors
    α_μ = pm.Normal("α_μ", mu=y_m.mean(), sigma=1)
    α_σ = pm.HalfNormal("α_σ", 5)
    β_μ = pm.Normal("β_μ", mu=0, sigma=1)
    β_σ = pm.HalfNormal("β_σ", sigma=5)

    # priors
    α = pm.Normal("α", mu=α_μ, sigma=α_σ, dims="group")
    β = pm.Normal("β", mu=β_μ, sigma=β_σ, dims="group")
    σ = pm.HalfNormal("σ", 5)
    _ = pm.Normal("y_pred", mu=α[idx] + β[idx] * x_m, sigma=σ, observed=y_m)

    idata_cen = pm.sample()
```

If you run `hierarchical_centered`, you will see a message from PyMC saying something like `There were 149 divergences after tuning. Increase target_accept or reparameterize.` This message means that samples generated from PyMC may not be trustworthy. So far, we have assumed that PyMC always returns samples that we can use without issues, but that's not always the case. In *Week 10*, we further discuss why this is, along with diagnostic methods to help you identify those situations and recommendations to fix the potential issues. In that section, we also explain what divergences are. For now, we will only say that when working with hierarchical linear models, we will usually get a lot of divergences.

The easy way to solve them is to increase `target_accept`, as PyMC kindly suggests. This is an argument of `pm.sample()` that defaults to 0.8 and can take a maximum value of 1. If you see divergences, setting this argument to values like 0.85, 0.9, or even higher can help. But if you reach values like 0.99 and still have divergences, you are probably out of luck with this simple trick and you need to do something else. And that's reparametrization. What is this? Reparametrization is writing a model in a different way, but that is mathematically equivalent to your original model: you are not changing the model, just writing it in another way. Many models, if not all, can be written in alternative ways. Sometimes, reparametrization can have a positive effect on the efficiency of the sampler or on the model's interpretability. For instance, you can remove divergences by doing a reparametrization. Let's see how to do that in the next section.

4.8.1 Centered vs. noncentered hierarchical models

There are two common parametrizations for hierarchical linear models, centered and non-centered. The `hierarchical_centered` model uses the centered one. The hallmark of this parametrization is that we are directly estimating parameters for individual groups; for instance, we are explicitly estimating the slope of each group. On the contrary, for the non-centered parametrization, we estimate the common slope for all groups and then a deflection for each group. It is important to notice that we are still modeling the slope of each group, but relative to the common slope, the information we are getting is the same, just represented differently. Since a model is worth a thousand words, let's check `hierarchical_non_centered`:

Code 4.14

```
with pm.Model(coords=coords) as hierarchical_non_centered:
    # hyperpriors
    α_μ = pm.Normal("α_μ", mu=y_m.mean(), sigma=1)
    α_σ = pm.HalfNormal("α_σ", 5)
    β_μ = pm.Normal("β_μ", mu=0, sigma=1)
    β_σ = pm.HalfNormal("β_σ", sigma=5)

    # priors
    α = pm.Normal("α", mu=α_μ, sigma=α_σ, dims="group")

    β_offset = pm.Normal("β_offset", mu=0, sigma=1, dims="group")
    β = pm.Deterministic("β", β_μ + β_offset * β_σ, dims="group")

    σ = pm.HalfNormal("σ", 5)
    _ = pm.Normal("y_pred", mu=α[idx] + β[idx] * x_m, sigma=σ, observed=y_m)

    idata_ncen = pm.sample(target_accept=0.85)
```

The difference is that for the model `hierarchical_centered`, we defined $\beta \sim \mathcal{N}(\beta_\mu, \beta_\sigma)$, and for `hierarchical_non_centered` we did $\beta = \beta_\mu + \beta_{\text{offset}} * \beta_\sigma$. The non-centered parametrization is more efficient: when I run the model I only get 2 divergences instead of 148 as before. To remove these remaining divergences, we may still need to increase `target_accept`. For this particular case, changing it from 0.8 to 0.85 worked like magic. To fully understand why this reparametrization works, you need to understand the geometry of the posterior distribution, but that's beyond the scope of this section. Don't worry, we will discuss this in *Week* 10.

Now that our samples are divergence-free, we can go back to analyze the posterior. *Figure* 4.17 shows the estimated values for α and β for `hierarchical_model`.

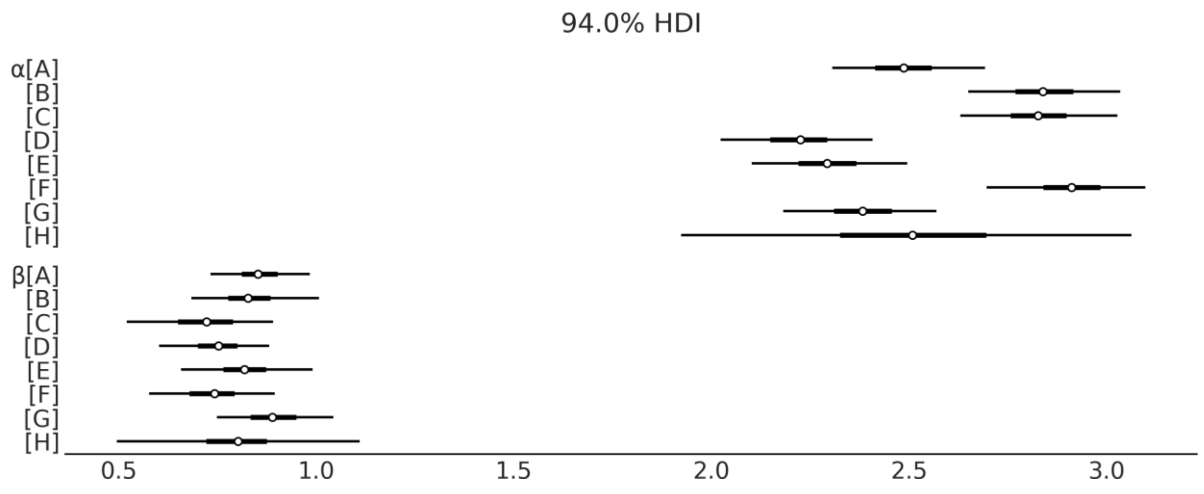

Figure 4.17: Posterior distribution for α and β for `hierarchical_non_centered`

The estimates for group `H` are still the ones with higher uncertainty. But the results look less crazy than those in *Figure* 4.16; the reason is that groups are sharing information. Hence, even when we don't have enough information to fit a line to a single point, group `H` *is being informed* by the other groups. Actually, all groups are informing all groups. This is the power of hierarchical models.

Figure 4.18 shows the fitted lines for each of the eight groups. We can see that we managed to fit a line to a single point. At first, this may sound weird or even fishy, but this is just a consequence of the structure of the hierarchical model. Each line is informed by the lines of the other groups, thus we are not truly adjusting a line to a single point. Instead, we are adjusting a line that's been informed by the points in the other groups to a single point.

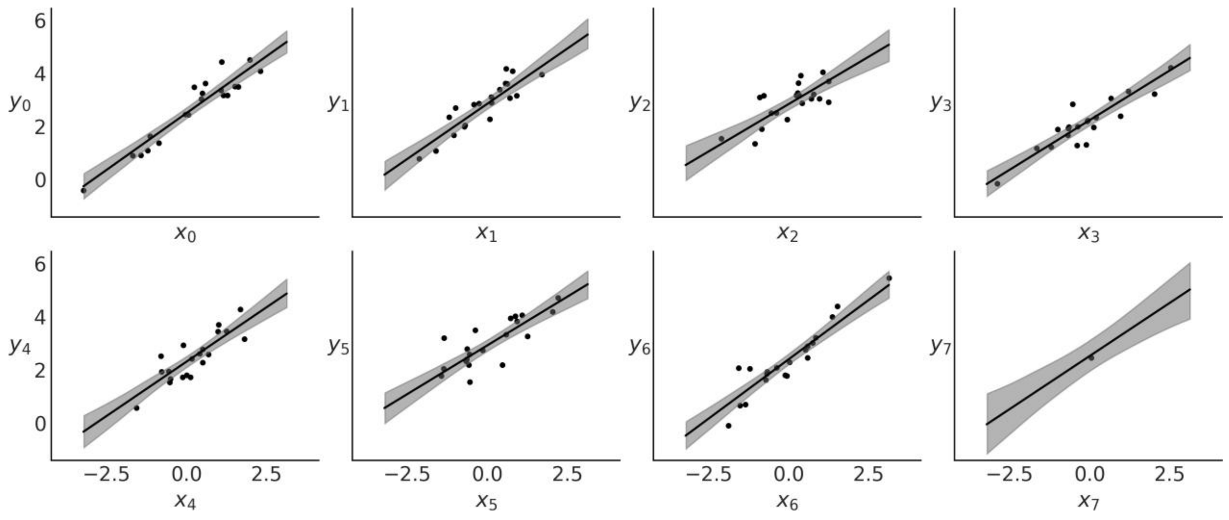

Figure 4.18: Fitted lines for `hierarchical_non_centered`

4.9 Multiple linear regression

So far, we have been working with one dependent variable and one independent variable. Nevertheless, it is not unusual to have several independent variables that we want to include in our model. Some examples could be:

- Perceived quality of wine (dependent) and acidity, density, alcohol level, residual sugar, and sulfates content (independent variables)
- A student's average grades (dependent) and family income, distance from home to school, and mother's education level (categorical variable)

We can easily extend the simple linear regression model to deal with more than one independent variable. We call this model multiple linear regression or, less often, multivariable linear regression (not to be confused with multivariate linear regression, the case where we have multiple dependent variables).

In a multiple linear regression model, we model the mean of the dependent variable as follows:

$$\mu = \alpha + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_k X_k$$

Using linear algebra notation, we can write a shorter version:

$$\mu = \alpha + \mathbf{X}\beta$$

$\mathbf{X}$ is a matrix of size $n \times k$ with the values of the independent variables, β is a vector of size k with the coefficients of the independent variables, and n is the number of observations.

If you are a little rusty with your linear algebra, you may want to check the Wikipedia article about the dot product between two vectors and its generalization to matrix multiplication: https://en.wikipedia.org/wiki/Dot_product. Basically, what you need to know is that we are just using a shorter and more convenient way to write our model:

$$\mathbf{X}\beta = \sum_{i}^{n} \beta_i X_i = \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_k X_k$$

Using the simple linear regression model, we find a straight line that (hopefully) explains our data. Under the multiple linear regression model, we find, instead, a hyperplane of dimension k. Thus, the multiple linear regression model is essentially the same as the simple linear regression model, the only difference being that now β is a vector and $\mathbf{X}$ is a matrix.

To see an example of a

multiple linear regression model, let’s go back to the bikes dataset. We will use the temperature and the humidity of the day to predict the number of rented bikes:

Code 4.15

```
with pm.Model() as model_mlb:
    α = pm.Normal("α", mu=0, sigma=1)
    β0  = pm.Normal("β0", mu=0, sigma=10)
    β1  = pm.Normal("β1", mu=0, sigma=10)
    σ = pm.HalfNormal("σ", 10)
    μ = pm.Deterministic("μ", pm.math.exp(α + β0 * bikes.temperature +
                                               β1 * bikes.hour))
    _ = pm.NegativeBinomial("y_pred", mu=μ, alpha=σ, observed=bikes.rented)

    idata_mlb = pm.sample()
```

Please take a moment to compare `model_mlb`, which has two independent variables, `temperature` and `hour`, with `model_neg`, which only has one independent variable, `temperature`. The only difference is that now we have two β coefficients, one for each independent variable. The rest of the model is the same. Notice that we could have written β = `pm.Normal("β1", mu=0, sigma=10, shape=2)` and then used `β1[0]` and `β1[1]` in the definition of μ. I usually do that.

As you can see, writing a multiple regression model is not that different from writing a simple regression model. Interpreting the results can be more challenging, though. For instance, the coefficient of `temperature` is now β_0 and the coefficient of `hour` is β_1. We can still interpret the coefficients as the change in the dependent variable for a unit change in the independent variable. But now we have to be careful to specify which independent variable we are talking about. For instance, we can say that for a unit increase in the temperature, the number of rented bikes increases by β_0 units, while holding the value of `hour` constant. Or we can say that for a unit increase in the hour, the number of rented bikes increases by β_1 units, while holding the value of `temperature` constant. Also, the value of a coefficient for a given variable is dependent on what other variables we are including in the model. For instance, the coefficient of `temperature` will vary depending on whether we incorporate the variable `hour` into the model or not.

Figure 4.19 shows the β coefficients for models `model_neg` (only `temperature`) and for model `model_mld` (`temperature` and `hour`).

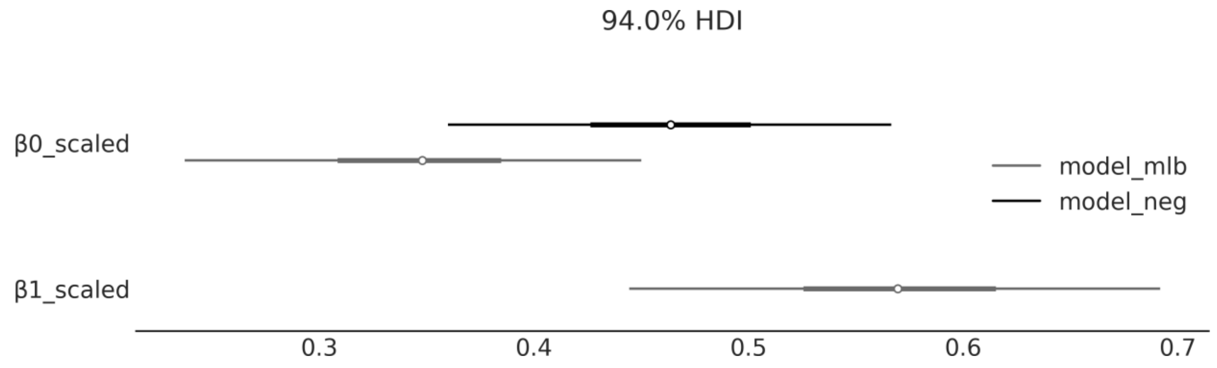

Figure 4.19: Scaled β coefficients for `model_neg` and `model_mlb`

We can see that the coefficient of `temperature` is different in both models. This is because the effect of `temperature` on the number of rented bikes depends on the hour of the day. Even more, the values of the β coefficients have been scaled by the standard deviation of their corresponding independent variable, so we can make them comparable. We can see that once we include `hour` in the model, the effect of `temperature` on the number of rented bikes gets smaller. This is because the effect of `hour` is already explaining some of the variations in the number of rented bikes that were previously explained by `temperature`. In extreme cases, the addition of a new variable can make the coefficient go to 0 or even change the sign. We will discuss more of this in the next Week.

4.10 Summary

In this Week, we have learned about linear regression, which aims to model the relationship between a dependent variable and an independent variable. We have seen how to use PyMC to fit a linear regression model and how to interpret the results and make plots that we can share with different audiences.

Our first example was a model with a Gaussian response. But then we saw that this is just one assumption and we can easily change it to deal with non-Gaussian responses, such as count data, using a NegativeBinomial regression model or a logistic regression model for binary data. We saw that when doing so we also need to set an inverse link function to map the linear predictor to the response variable. Using a Student's t-distribution as the likelihood can be useful for dealing with outliers. We spent most of the Week modeling the mean as a linear function of the independent variable, but we learned that we can also model other parameters, like the variance. This is useful when we have heteroscedastic data. We learned how to apply the concept of partial pooling to create hierarchical linear regression models. Finally, we briefly discussed multiple linear regression models.

PyMC makes it very easy to implement all these different flavors of Bayesian linear regression by changing one or a few lines of code. In the next Week, we will learn more about linear regression and we will learn about Bambi, a tool built on top of PyMC that makes it even easier to build and analyze linear regression models.

4.11 ASSIGNMENT:

1. Using the howell dataset, create a linear model of the weight (x) against the height (y). Exclude subjects that are younger than 18. Explain the results.
2. For four subjects, we get the weights (45.73, 65.8, 54.2, 32.59), but not their heights. Using the model from the previous exercise, predict the height for each subject, together with their 50% and 94% HDIs. Tip: Use `pm.MutableData`.
3. Choose a dataset that you find interesting and use it with the simple linear regression model. Be sure to explore the results using ArviZ functions. If you do not have an interesting dataset, try searching online, for example, at http://data.worldbank.org or http://www.stat.ufl.edu/~winner/datasets.html.