# Generalizing Linear Models

Min-Te Sun, Ph.D.

1

## Generalize Linear Model to Many Problems

- In the last chapter, we used a linear combination of input variables to predict the mean of an output variable. We assumed the latter to be distributed as a Gaussian. Using a Gaussian works in many situations, but for many other it could be wiser to choose a different distribution; we already saw an example of this when we replaced the Gaussian distribution with a Student's t-distribution. In this chapter, we will see more examples where it is wise to use distributions other than Gaussian. As we will learn, there is a general motif, or pattern, that can be used to generalize the linear model to many problems.

2

## Generalized Linear Models

- One of the core ideas of this chapter is rather simple: in order to predict the mean of an output variable, we can apply an arbitrary function to a linear combination of input variable.

$$\mu = f(\alpha + X\beta)$$

- Where $f$ is a function, we will call inverse link function. There are many inverse link functions we can choose; probably the simplest one is the identity function. This is a function that returns the same value used as its argument. All models from "Modeling with Linear Regression" used the identity function, and for simplicity we just omit it. The identity function may not be very useful on its own, but it allows us to think of several different models in a more unified way.
- Why do we call $f$ the inverse link function, instead of just the link function? Because traditionally people apply functions to the other side of the equation, and unfortunately for us, they already called dibs on the term link function—so to avoid confusion, we are going to stick to the term inverse link function.

3

## Why Inverse Link Function?

- One situation under which we would like to use an inverse link function is when working with categorical variables, such as color names, gender, biological species, or political party/affiliation. None of these variable is well-modeled by Gaussians. Think about it, in principle, a Gaussian works well for a continuous variable taking any value on the real line, while the variables mentioned here are discrete and only take a few values (such as red, green, or blue). If we change the distribution we used to model the data, we will in general need to also change how we model the plausible values for the mean of those distributions. For example, if we use a binomial distribution, like in "*Thinking Probabilistically*" and "*Programming Probabilistically*", we will need a linear model that returns a mean value in the [0, 1] interval; one way to achieve this is to keep the linear model but use an inverse link function to restrict the output to the desired interval. This trick is not restricted to discrete variables; we may want to model data that can only take positive values, and thus we may want to restrict the linear model to return positive values for the mean of a distribution, such as Gamma or exponential.

4

## Quantitative or Qualitative Variables?

- Just before moving on, note that some variables can be codified as quantitative or as qualitative, and that this is a decision you have to make based on the context of your problem; for example, we can talk about the red and green categorical variables if we are talking about color names, or we can talk about the 650 nm and 510 nm continuous variables if we are working with wavelengths.

5

## Logistic Regression

- Regression problems are about predicting a continuous value for an output variable given the values of one or more input variables. Instead, classification is about assigning a discrete value (representing a discrete class) to an output variable given some input variables. In both cases, the task is to get a model that properly models the mapping between output and input variables; in order to do so, we have at our disposal a sample with *correct* pairs of output-input variables. From a **machine learning** perspective, both regressions and classifications are instances of supervised learning algorithms.
- My mother prepares a delicious dish called **sopa seca**, which is basically a spaghetti-based recipe and literally means *dry soup*. While it may sound like a misnomer or even an oxymoron, the name of the dish makes total sense when we learn how it is cooked. Something similar happens with logistic regression, a model that, despite its name, is generally framed as a method to solve classification problems.

6

## Logistic Function as Inverse Link Function

- The logistic regression model is a generalization of the linear regression model from "Modeling with Linear Regression", and thus its name. We achieve this generalization by replacing $f$ in slide #3 with the logistic function as an inverse link function:

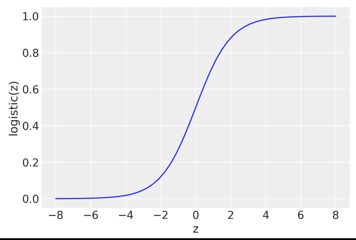
$$logistic(z) = \frac{1}{1 + e^{-z}}$$

- 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 by executing the next few lines:

```
z = np.linspace(-8, 8)
plt.plot(z, 1 / (1 + np.exp(-z)))
plt.xlabel('z')
plt.ylabel('logistic(z)')
```

7

## The Plot of Logistic (Sigmoid) Function



8

## 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 or instances, 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 similar to the coin-flipping example from "Programming Probabilistically" and "Modeling with Linear Regression".
- We may remember we used the Bernoulli distribution as the likelihood. The difference with the coin-flipping problem is that now $\theta$ is not going to be generated from a beta distribution; instead, $\theta$ is going to be defined by a linear model with the logistic as the inverse link function. Omitting the priors we have:

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

- Notice that the main difference with the simple linear regression from *"Modeling with Linear Regression"* is the use of a Bernoulli distribution instead of a Gaussian distribution, and the use of the logistic function instead of the identity function.

9

## The Iris Dataset

- We are going to apply logistic regression to the iris dataset. This is a classic dataset containing information about flowers from three closely related species: setosa, virginica, and versicolor. These are going to be our dependent variables, that is, the classes we want to predict. We have 50 individual cases of each species and for each individual case, the dataset contains four variables that we are going to use as the independent variables (or features): 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 can load a data frame with the iris dataset by doing the following:

```
iris = pd.read_csv('../data/iris.csv')
iris.head()
```

10

## First Few Entries of Iris Dataset

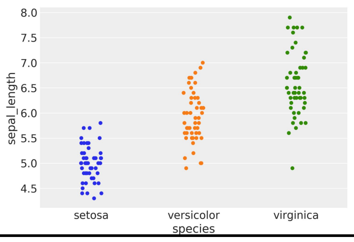
| | sepal_length | sepal_width | petal_length | petal_width | species |
|---|---|---|---|---|---|
| 0 | 5.1 | 3.5 | 1.4 | 0.2 | setosa |
| 1 | 4.9 | 3.0 | 1.4 | 0.2 | setosa |
| 2 | 4.7 | 3.2 | 1.3 | 0.2 | setosa |
| 3 | 4.6 | 3.1 | 1.5 | 0.2 | setosa |
| 4 | 5.0 | 3.6 | 1.4 | 0.2 | setosa |

- Now, we will plot the three species versus sepal_length using the stripplot function from seaborn:

```
sns.stripplot(x="species", y="sepal_length", data=iris, jitter=True)
```

11

## The Plot



12

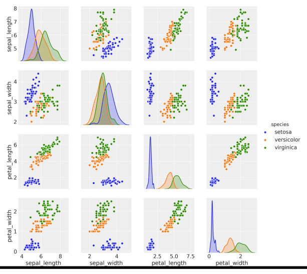
## Plot Explanation

- In the figure in the previous slide, the *y* axis is continuous while the *x* axis is categorical; the dispersion (or jitter) of the points along the *x* axis has no meaning at all, it is just a trick we add with the jitter argument to avoid having all the points collapsed onto a single line. (Try setting the jitter argument to False to see what it means.) The only thing that matters when reading the *x* axis is the membership of the points to the setosa, versicolor, or virginica classes. You may also try other plots for this data, such as violin plots, which are also available as one-liners with seaborn.
- Another way to inspect the data is by doing a scatter matrix with the pairplot function. We have scatter plots arranged in a 4 x 4 grid, since we have four features in the iris dataset. The grid is symmetrical, with the upper and lower triangles showing the same information. The scatter plot on the main diagonal should correspond to a variable against itself; given that such a plot is not informative at all, we have replaced those scatters plots with a kde for each feature. Inside each subplot, we have the three species (or classes) represented with a different color, which is the same as used in the figure in the previous slide:

```
sns.pairplot(iris, hue='species', diag_kind='kde')
```

13

## How Features and Classes are Related in Iris Dataset



14

## Logistic Model Applied to Iris Dataset

- We are going to begin with the simplest possible classification problem: two classes, setosa and versicolor, and just one independent variable or feature, the sepal_length. As it is usually done, we are going to encode the setosa and versicolor categorical variables with the numbers 0 and 1. Using pandas, we can do the following:

```
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 proper format, we can finally build the model with PyMC3.

15

## First Part of Model_0 Resembles Linear Regression Model

- Notice how the first part of model_0 resembles a linear regression model. Also pay attention to the two deterministic variables: θ and bd. θ is the output of the logistic function applied to the μ variable, and bd is the boundary decision, which is the value used to separate classes; we will discuss this later in detail. Another point worth mentioning is that instead of explicitly writing the logistic function, we are using pm.math.sigmoid (this is just an alias for the Theano function with the same name):

```
with pm.Model() as model_0:
        α = pm.Normal('α', mu=0, sd=10)
        β = pm.Normal('β', mu=0, sd=10)
        μ = α + pm.math.dot(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)

trace_0 = pm.sample(1000)
```

16

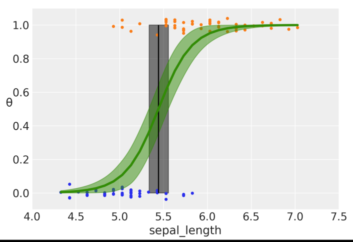
## Code to Generate Plot

- In order to save pages and avoid you getting bored with the same type of plot over and over again, we are going to omit doing a trace plot and other similar summaries, but we encourage you make your own plots and summaries to further explore the examples in the textbook. Instead, we are going to jump directly to generating the figure in the next slide, a plot of the data, together with the fitted sigmoid curve and the decision boundary:

```
theta = trace_0['θ'].mean(axis=0)
idx = np.argsort(x_c)
plt.plot(x_c[idx], theta[idx], color='C2', lw=3)
plt.vlines(trace_0['bd'].mean(), 0, 1, color='k')
bd_hpd = az.hpd(trace_0['bd'])
plt.fill_betweenx([0, 1], bd_hpd[0], bd_hpd[1], color='k', alpha=0.5)
plt.scatter(x_c, np.random.normal(y_0, 0.02), marker='.', color=[f'C{x}' for x in y_0])
az.plot_hpd(x_c, trace_0['θ'], color='C2')
plt.xlabel(x_n
plt.ylabel('θ', rotation=0)
# use original scale for xticks
locs, _ = plt.xticks()
plt.xticks(locs, np.round(locs + x_0.mean(), 1))
```

17

## The Plot



18

## Plot Explanation

- The figure in previous slide shows the sepal length versus the iris species (setosa = 0, versicolor = 1). To avoid over-plotting, the binary response variables are jittered. An S-shaped (green) line is the mean value of $\theta$. 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 (green) band is the 94% HPD interval. The boundary decision is represented as a (black) vertical line with a semi-transparent band for its 94% HPD. According to the boundary decision, the $x_i$ values (sepal lengths, in this case) to the left correspond to class 0 (setosa), and the values to the right to class 1 (versicolor).

19

## Decision Boundary

- The decision boundary is defined as the value of $x_i$, for which $y = 0.5$. And it turns out to be $-\frac{\alpha}{\beta}$, which we can derive as follows:
- From the definition of the model, we have the following relationship:
$$\theta = logistic(\alpha + x\beta)$$
- And from the definition of the logistic function, we have $\theta = 0.5$ when the argument of the logistic regression is 0:
$$0.5 = logistic(\alpha + x_i\beta) \Leftrightarrow 0 = \alpha + x_i\beta$$
- By reordering the previous equation, we find that the value of $x_i$, for which $\theta = 0.5$, corresponds to the following expression:
$$x_i = -\frac{\alpha}{\beta}$$

20

## Key Points

- The value of $\theta$ is, generally speaking, $p(y = 1|x)$. In this sense, the 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, that is, a number in the [0-1] interval. Then, we introduce a rule to turn this probability into a two-class assignment. In this case, if $p(y = 1) \geq 0.5$, we assign class 1, otherwise 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. We may argue this boundary is only reasonable if we are OK making a mistake in either one direction or the other—in other words, if it is the same for us to misclassify a setosa as a versicolor or a versicolor as a setosa. It turns out that this is not always the case, and the cost associated with the miss-classification does not need to be symmetrical, as you may remember from *"Programming Probabilistically"*, when we discussed loss functions.

21

## Multiple Logistic Regression

- In a similar fashion to multiple linear regression, multiple logistic regression is about using more than one independent variable. Let's try combining the sepal length and the sepal width. Remember we need to pre-process the data a little bit:

```
df = iris.query("species == ('setosa', 'versicolor')")
y_1 = pd.Categorical(df['species']).codes
x_n = ['sepal_length', 'sepal_width']
x_1 = df[x_n].values
```

22

## The Boundary Decision

- From the model, we have the following equation:
$$\theta = logistic(\alpha + \beta_1 x_1 + \beta_2 x_2)$$
- And from the definition of the logistic function, we have $\theta = 0.5$, when the argument of the logistic regression is zero:
$$0.5 = logistic(\alpha + \beta_1 x_1 + \beta_2 x_2) \Leftrightarrow 0 = \alpha + \beta_1 x_1 + \beta_2 x_2$$
- By reordering, we find the value of $x_2$ for which $\theta = 0.5$ corresponds to the following expression:
$$x_2 = -\frac{\alpha}{\beta_2} + \left(-\frac{\beta_1}{\beta_2} x_1\right)$$

23

## Hyperplane as Boundary

- This expression for the boundary decision has the same mathematical form as a line equation, with the first term being the intercept and the second the slope. The parentheses are used for clarity and we can omit them if we want. The boundary being a line is totally reasonable, isn't it? If we have one feature, we have unidimensional data and we can split it into two groups using a point; if we have two features, we have a two-dimensional data space and we can separate it using a line; for three dimensions, the boundary will be a plane; and for higher dimensions, we will talk generically about hyperplanes. In fact, a hyperplane is a general concept defined roughly as the subspace of dimension *n* - 1 of an *n*-dimensional space, so we can always talk about hyperplanes!

24

## Implementing The Model

- To write the multiple logistic regression model using PyMC3, we take advantages of its vectorization capabilities, allowing us to introduce only minor modifications to the previous simple logistic model (model_0):

```
with pm.Model() as model_1:
          α = pm.Normal('α', mu=0, sd=10)
          β = pm.Normal('β', mu=0, sd=2, shape=len(x_n))
          μ = α + pm.math.dot(x_1, β)
          θ = pm.Deterministic('θ', 1 / (1 + pm.math.exp(-μ)))
          bd = pm.Deterministic('bd', -α/β[1] - β[0]/β[1] * x_1[:,0])
          yl = pm.Bernoulli('yl', p=θ, observed=y_1)
trace_1 = pm.sample(2000)
```

25

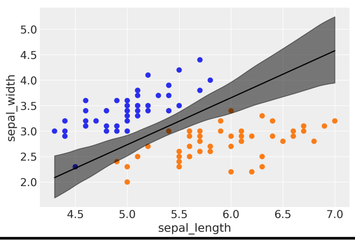
## Code to Plot Data

- As we did for a single predictor variable, we are going to plot the data and the decision boundary:

```
idx = np.argsort(x_1[:,0])
bd = trace_1['bd'].mean(0)[idx]
plt.scatter(x_1[:,0], x_1[:,1], c=[f'C{x}' for x in y_0])
plt.plot(x_1[:,0][idx], bd, color='k');
az.plot_hpd(x_1[:,0], trace_1['bd'], color='k')
plt.xlabel(x_n[0])
plt.ylabel(x_n[1])
```

26

## The Plot



27

## Plot Explanation

- The boundary decision is a straight line, as we have already seen. Do not get confused by the curved aspect of the 94% HPD band. The apparent curvature is the result of having multiple lines pivoting around a central region (roughly around the mean of $x$ and the mean of $y$).

28

## Interpreting Coefficients of Logistic Regression

- We must be careful when interpreting the $\beta$ coefficients of a logistic regression. Interpretation is not as straightforward as with linear models, which we looked at in "Modeling with Linear Regression". Using the logistic inverse link function introduces a non-linearity that we have to take into account. If $\beta$ is positive, increasing $x$ will increase $p(y = 1)$ by some amount, but the amount is not a linear function of $x$; instead, it depends non-linearly on the value of $x$. We can visualize this fact in the figure in slide #18; 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 $\beta$.

29

## Little Bit of Algebra

- The basic logistic model is:
$$\theta = logistic(\alpha + X\beta)$$
- The inverse of the logistic is the logit function, which is:
$$logit(z) = \log(\frac{z}{1-z})$$
- Thus, if we take the first equation in this section and apply the logit function to both terms, we get this equation:
$$logit(\theta) = \alpha + X\beta$$
- Or equivalently:
$$\log\left(\frac{\theta}{1-\theta}\right) = \alpha + X\beta$$

30

## Odds

- Remember that $\theta$ in our model is $p(y=1)$:
$$\log\left(\frac{p(y=1)}{1-p(y=1)}\right)=\alpha+X\beta$$
- The $\frac{p(y=1)}{1-p(y=1)}$quantity is known as the **odds**.
- The odds of success are defined as the ratio of the probability of success over the probability of not-success. While the probability of getting 2 by rolling a fair die is 1/6, the odds for the same event are $\frac{1/6}{5/6}$= 0.2 or one favorable event to five unfavorable events. Odds are often used by gamblers mainly because odds provide a more intuitive tool than raw probabilities when thinking about the proper way to bet.
- In a logistic regression, the $\beta$ coefficient encodes the increase in log-odds units by unit increase of the $x$ variable.

31

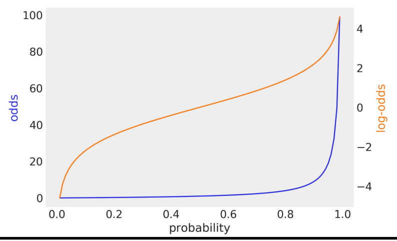
## Transformation from Probability to Odds

- 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. The figure in the next slide shows how probabilities are related to odds and log-odds:

```
probability = np.linspace(0.01, 1, 100)
odds = probability / (1 - probability)
_, ax1 = plt.subplots()
ax2 = ax1.twinx()
ax1.plot(probability, odds, 'C0’)
ax2.plot(probability, np.log(odds), 'C1’)
ax1.set_xlabel('probability’)
ax1.set_ylabel('odds', color='C0’)
ax2.set_ylabel('log-odds', color='C1’)
ax1.grid(False)
ax2.grid(False)
```

32

## The Plot



33

## Coefficients in Summary Are Log-Odds Scale

- Thus, the values of the coefficients provided by summary are in the log-odds scale:

```
df = az.summary(trace_1, var_names=varnames)
df
```

| | **mean** | **sd** | **mc error** | **hpd 3%** | **hpd 97%** | **eff_n** | **r_hat** |
|---|---|---|---|---|---|---|---|
| **α** | -9.12 | 4.61 | 0.15 | -17.55 | -0.42 | 1353.0 | 1.0 |
| **β[0]** | 4.65 | 0.87 | 0.03 | 2.96 | 6.15 | 1268.0 | 1.0 |
| **β[1]** | -5.16 | 0.95 | 0.01 | -7.05 | -3.46 | 1606.0 | 1.0 |

34

## One Way of Understanding Models

- One very pragmatic way of understanding models is to change parameters and see what happens. In the following block of code, we are computing the log-odds in favor of versicolor as $\text{log\_odds\_versicolor\_i} = \alpha + \beta_1 x_1 + \beta_2 x_2$, and then the probability of versicolor with the logistic function. Then, we repeat the computation by fixing $x_2$ and increasing $x_1$by 1:

```
x_1 = 4.5  # sepal_length
x_2 = 3   # sepal_width
log_odds_versicolor_i = (df['mean'] * [1, x_1, x_2]).sum()
probability_versicolor_i = logistic(log_odds_versicolor_i)
log_odds_versicolor_f = (df['mean'] * [1, x_1 + 1, x_2]).sum()
probability_versicolor_f = logistic(log_odds_versicolor_f)
log_odds_versicolor_f - log_odds_versicolor_i, probability_versicolor_f –
probability_versicolor_i
```

35

## Increase in Log-Odds ≈ $\beta_0$

- If you run the code, you will find that the increase in log-odds is ≈ 4.66, which is exactly the value of $\beta_0$ (check the summary of trace_1). This is in line with our previous finding that encodes the increase in log-odds units by unit increase of the $x$ variable. The increase in probability is ≈ 0.70.

36

## Dealing with Correlated Variables

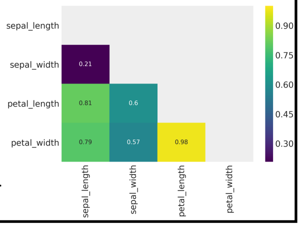
- We know from *"Modeling with Linear Regression"* that tricky things await us when we deal with (highly) correlated variables. Correlated variables translate into wider combinations of coefficients that are able to explain the data, or from a complementary point of view, correlated data has less power to restrict the model. A similar problem occurs when the classes become perfectly separable, that is, when there is no overlap between classes given the linear combination of variables in our model.
- Using the iris dataset, you can try running model_1, but this time using the petal_width and petal_length variables. You will find that the $\beta$ coefficients are broader than before, and also the 94% HPD black band in the figure in slide #27 is much wider:

```
corr = iris[iris['species'] != 'virginica'].corr()
mask = np.tri(*corr.shape).T
sns.heatmap(corr.abs(), mask=mask, annot=True, cmap='viridis')
```

37

## Heat Map

- The figure on the right is a heat map showing that for the sepal_length and setal_width variables used in the first example, the correlation is not as high as the correlation between the petal_length and petal_width variables used in the second example:
- To generate the figure on the right, we have used a mask to remove the upper triangle and the diagonal elements of the heat map, since these are uninformative, given the lower triangle. Also note that we have plotted the absolute value of the correlation, since at this moment we do not care about the sign of the correlation between variables, only about its strength.



38

## Weakly-Informative Priors Suggested by Andrew Gelman

- One solution when dealing with (highly) correlated variables is to just remove one (or more than one) correlated variable. Another option is to put more information into the prior, this can be achieved using informative priors if we have useful information. For weakly-informative priors, Andrew Gelman and the Stan Team recommend (https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations) scaling all non-binary variables to have a mean of 0 and then using:

$$\beta \sim StudentT(0, \nu, sd)$$

- Here, $sd$ should be chosen in order to weakly inform us about the expected values for the scale. The normality parameter, $\nu$, is suggested to be around 3-7. This prior is saying that, in general, we expect the coefficient to be small, but we use fat tails because occasionally we will find some larger coefficients. As we saw in "Programming Probabilistically" and "Modeling with Linear Regression", using a Student's t-distribution leads to a more robust model than using a Gaussian distribution.

39

## Dealing with Unbalanced Classes

- The iris dataset is completely balanced, in the sense that each category has exactly the same number of observations. We have 50 setosas, 50 versicolors, and 50 virgininas. This is something to thank Ronald Fisher for, unlike his dedication to popularizing the use of p-values.
- On the contrary, many datasets consist of unbalanced data, that is, there are many more data points from one class than from another. When this happens, logistic regression can run into trouble, namely the boundary cannot be determined as accurately as when the dataset is more balanced.

40

## Example of Unbalanced Classes

- To see an example of this behavior, we are going to use the iris dataset and we are going to arbitrarily remove some data points from the setosa class:

```
df = iris.query("species == ('setosa', 'versicolor')")
df = df[45:]
y_3 = pd.Categorical(df['species']).codes
x_n = ['sepal_length', 'sepal_width’]
x_3 = df[x_n].values
```

- And then, we are going to run a multiple logistic regression model, just as before:

```
with pm.Model() as model_3:
        α = pm.Normal('α', mu=0, sd=10)
        β = pm.Normal('β', mu=0, sd=2, shape=len(x_n))
        μ = α + pm.math.dot(x_3, β)
        θ = 1 / (1 + pm.math.exp(-μ))
        bd = pm.Deterministic('bd', -α/β[1] - β[0]/β[1] * x_3[:,0])
        yl = pm.Bernoulli('yl', p=θ, observed=y_3)
trace_3 = pm.sample(1000)
```

41

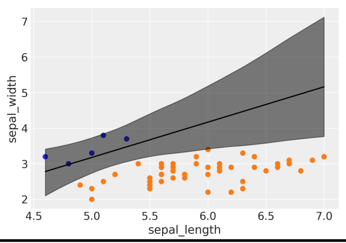
## Boundary Shifted Toward Less Abundant Class

- As we can see from the figure in the next slide, the boundary decision is now shifted toward the less abundant class, and the uncertainty is larger than before. This is the typical behavior of a logistic model for unbalanced data. But wait a minute! You may be arguing that I am cheating here since the wider uncertainty could be the product of having less total data and not just fewer setosas than versicolors! That could be a valid point; try with *exercise 6* to verify that what explains this plot is the unbalanced data:

```
idx = np.argsort(x_3[:,0])
bd = trace_3['bd'].mean(0)[idx]
plt.scatter(x_3[:,0], x_3[:,1], c= [f'C{x}' for x in y_3])
plt.plot(x_3[:,0][idx], bd, color=’k’)
az.plot_hpd(x_3[:,0], trace_3['bd'], color='k’)
plt.xlabel(x_n[0])
plt.ylabel(x_n[1])
```

42

## The Plot



43

## What To Do With Unbalanced Data?

- What do we do if we find unbalanced data? Well, the obvious solution is to get a dataset with roughly the same data points per class. This can be important to have in mind if you are collecting or generating the data. If you have no control over the dataset, you should be careful when interpreting the result for unbalanced data. Check the uncertainty of the model and run some posterior predictive checks to see whether the results are useful to you. Another option will be to input more prior information, if available, and/or run an alternative model, as explained later in this set of slides.

44

## Softmax Regression

- One way to generalize logistic regression to more than two classes is with **softmax regression**. We need to introduce two changes with respect to logistic regression; first, we replace the logistic function with the softmax function:

$$softmax_i(\mu) = \frac{\exp(\mu_i)}{\sum \exp(\mu_k)}$$

- In other words, to obtain the output of the softmax function for the i-esim element of a vector, $\mu$, we take the exponential of the i-esim value divided by the sum of all the exponentiated values in the $\mu$ vector.

45

## Boltzmann Distribution

- Softmax guarantees we will get positive values that sum up to 1. The softmax function is reduced to the logistic function when $k = 2$. As a side note, the softmax function has the same form as the **Boltzmann distribution** used in statistical mechanics, which is a very powerful branch of physics dealing with the probabilistic description of atomic and molecular systems. The Boltzmann distribution (and softmax, in some fields) has a parameter called temperature, $T$, that divides $\mu$; when $T \rightarrow \infty$, the probability distribution becomes flat and all states are equally likely, and when $T \rightarrow 0$, only the most probable state gets populated, and thus softmax behaves like a max function.

46

## Replace Bernoulli Distribution with Categorical Distribution

- The second modification is that we replace the Bernoulli distribution with the categorical distribution. The categorical distribution is the generalization of the Bernoulli to more than two outcomes. Also, as the Bernoulli distribution (single coin flip) is a special case of the Binomial ($n$ coin flips), the categorical (single roll of a die) is a special case of the multinomial distribution ($n$ rolls of a die).

47

## Use All Three Classes in Iris Dataset

- To exemplify the softmax regression, we are going to continue working with the iris dataset, only this time we are going to use its three classes (setosa, versicolor, and virginica) and its four features (sepal length, sepal width, petal length, and petal width). We are also going to standardize the data, since this will help the sampler to run more efficiently (we could have also just centered the data):

```
iris = sns.load_dataset('iris')
y_s = pd.Categorical(iris['species']).codes
x_n = iris.columns[:-1]
x_s = iris[x_n].values
x_s = (x_s - x_s.mean(axis=0)) / x_s.std(axis=0)
```

48

## Code Changes between Logistic and Softmax Models

- The PyMC3 code reflects the changes between the logistic and softmax models. Notice the shapes of the $\alpha$ and $\beta$ coefficients. We use the softmax function from Theano; we have used the import theano.tensor as tt idiom, which is the convention used by PyMC3 developers:

```
with pm.Model() as model_s:
        α = pm.Normal('α', mu=0, sd=5, shape=3)
        β = pm.Normal('β', mu=0, sd=5, shape=(4,3))
        μ = pm.Deterministic('μ', α + pm.math.dot(x_s, β))
        θ = tt.nnet.softmax(μ)
        yl = pm.Categorical('yl', p=θ, observed=y_s)
trace_s = pm.sample(2000)
```

49

## How Well Does Our Model Perform?

- Let's find out by checking how many cases we can predict correctly. In the following code, we just use the mean of the parameters to compute the probability of each data point belonging to each of the three classes, then we assign the class by using the argmax function. And we compare the result with the observed values:

```
data_pred = trace_s['μ'].mean(0)
y_pred = [np.exp(point)/np.sum(np.exp(point), axis=0)
for point in data_pred]
        f'{np.sum(y_s == np.argmax(y_pred, axis=1)) / len(y_s):.2f}'
```

50

## 98% Correct Classification

- The result is that we classify $\sim 98\%$ of the data points correctly, that is, we miss only three cases. This is very good. Nevertheless, a true test to evaluate the performance of our model will be to check it on data not used to fit the model. Otherwise, we may be overestimating the ability of the model to generalize to other data. We will discuss this subject in detail in “Modeling with Linear Regression”. For now, we will leave this just as an auto-consistency test indicating that the model runs OK.

51

## Marginal Distributions of Parameter Are Wide

- You may have noticed that the posterior, or more properly the marginal distributions of each parameter, are very wide; in fact, they are as wide as indicated by the priors. Even when we were able to make correct predictions, this does not look OK. This is the same non-identifiability problem we already encountered for correlated data in other regression models or with perfectly separable classes. In this case, the wide posterior is due to the condition that all probabilities must sum to one. Given this condition, we are using more parameters than we need to fully specify the model. In simple terms, if you have ten numbers that sum to one, you just need to give me nine of them; the other I can compute. One solution is to fix the extra parameters to some value, for example, zero. The code in the next slide shows how to achieve this using PyMC3:

52

## Code to Fix Extra Parameters

```
with pm.Model() as model_sf:
        α = pm.Normal('α', mu=0, sd=2, shape=2)
        β = pm.Normal('β', mu=0, sd=2, shape=(4,2))
        α_f = tt.concatenate([[0] , α])
        β_f = tt.concatenate([np.zeros((4,1)) , β], axis=1)
        μ = α_f + pm.math.dot(x_s, β_f)
        θ = tt.nnet.softmax(μ)
        yl = pm.Categorical('yl', p=θ, observed=y_s)
trace_sf = pm.sample(1000)
```

53

## Use Threshold to Turn *Continuous* Computed Probability into *Discrete* Boundary

- So far, we have discussed logistic regression and a few extensions of it. In all cases, we tried to directly compute $p(y|x)$, that is, the probability of a given class knowing $x$, with $x$ being feature(s) we measured to members of the classes. In other words, we try to directly model the mapping from the independent variables to the dependent ones, and then use a threshold to turn the *continuous* computed probability into a *discrete* boundary that allows us to assign classes.

54

## Generative Classifier vs Discriminative Classifier

- This approach is not unique. One alternative is to first model $p(x|y)$, that is, the distribution of $x$ for each class, and then assign the classes. This kind of model is called a **generative classifier**, because we are creating a model from which we can *generate* samples from each class. On the other hand, logistic regression is a type of **discriminative classifier**, since it tries to classify by *discriminating* classes but we cannot generate examples from each class from the model.
  - We are not going to go into much detail here about generative models for classification, but we are going to see one example that illustrates the core of this type of model for classification. We are going to do it for two classes and only one feature, exactly as the first model we built in this chapter (model_0), and we are going to use the very same data.

55

## Code of Generative Classifier

- The following is a PyMC3 implementation of a generative classifier. From the code, you can see that now the boundary decision is defined as the average between the estimated Gaussian means. This is the correct boundary decision when the distributions are normal and their standard deviations are equal. These are the assumptions made by a model known as **linear discriminant analysis** (**LDA**). Despite its name, the LDA model is generative:

```
with pm.Model() as lda:
        μ = pm.Normal('μ', mu=0, sd=10, shape=2)
        σ = pm.HalfNormal('σ', 10)
        setosa = pm.Normal('setosa', mu=μ[0], sd=σ, observed=x_0[:50])
        versicolor = pm.Normal('versicolor', mu=μ[1], sd=σ, observed=x_0[50:])
        bd = pm.Deterministic('bd', (μ[0] + μ[1]) / 2)
trace_lda = pm.sample(1000)
```

56

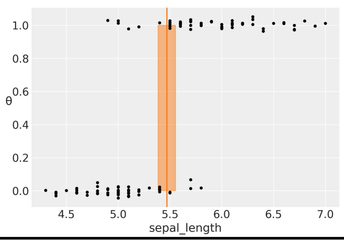
## Code to Plot Two Classes

- Now, we are going to plot a figure showing the two classes (setosa = 0 and versicolor = 1) against the values for sepal length, and also the boundary decision as a red line and the 94% **Highest-Posterior Density** (**HPD**) interval for it as a semitransparent red band:

```
plt.axvline(trace_lda['bd'].mean(), ymax=1, color='C1’)
bd_hpd = az.hpd(trace_lda['bd’])
plt.fill_betweenx([0, 1], bd_hpd[0], bd_hpd[1], color='C1', alpha=0.5)
plt.plot(x_0, np.random.normal(y_0, 0.02), '.', color='k’)
plt.ylabel('θ', rotation=0)
plt.xlabel('sepal_length')
```

57

## The Plot



58

## Plot Explanation

- Compare the figure in previous slide with the figure in slide #18; they're pretty similar, right? Also, check the boundary decision in the following summary:

```
az.summary(trace_lda)
```

| | mean | sd | mc error | hpd 3% | hpd 97% | eff_n | r_hat |
|---|---|---|---|---|---|---|---|
| **μ[0]** | 5.01 | 0.06 | 0.0 | 4.89 | 5.13 | 2664.0 | 1.0 |
| **μ[1]** | 5.94 | 0.06 | 0.0 | 5.82 | 6.06 | 2851.0 | 1.0 |
| **σ** | 0.45 | 0.03 | 0.0 | 0.39 | 0.51 | 2702.0 | 1.0 |
| **bd** | 5.47 | 0.05 | 0.0 | 5.39 | 5.55 | 2677.0 | 1.0 |

59

## Quadratic Linear Discriminant

- Both the LDA model and the logistic regression provide similar results. The linear-discriminant model can be extended to more than one feature by modeling the classes as multivariate Gaussians. Also, it is possible to relax the assumption of the classes sharing a common variance (or covariance). This leads to a model known as **quadratic linear discriminant** (**QDA**), since now the decision boundary is not linear but quadratic.
- In general, LDA or QDA models will work better than logistic regression when the features we are using are more or less Gaussian-distributed, and logistic regression will perform better in the opposite case. One advantage of the generative model for classification is that it may be easier or more natural to incorporate prior information; for example, we may have information about the mean and variance of the data to incorporate into the model.

60

## Boundary Decisions of LDA and QDA Are Closed Form

- It is important to note that the boundary decisions of LDA and QDA are known in closed form and hence they are usually computed in such a way. To use an LDA for two classes and one feature, we just need to compute the mean of each distribution and average those two values, and we get the boundary decision. In the preceding model, we just did that but with a Bayesian twist. We estimated the parameters of the two Gaussians and then we plugged those estimates into a predefined formula. Where do such formulas come from?
  - Without getting into too much detail, to obtain that formula, we must assume that the data is Gaussian-distributed, and hence such a formula will only work if the data does not deviate drastically from normality. Of course, we may hit a problem if we want to relax the normality assumption, such as by using a Student's t-distribution (or a multivariate Student's t-distribution, or whatever). In such a case, we can no longer use the closed form for the LDA (or QDA); nevertheless, we can still compute a decision boundary numerically using PyMC3.

61

## Poisson Regression (for Count Data)

- Another very popular generalized linear model is the Poisson regression. This model assumes data is distributed according to the Poisson distribution.
- One scenario where Poisson distribution is useful is when counting things, such as the decay of a radioactive nucleus, the number of children per couple, or the number of Twitter followers. What all these examples have in common is that we usually model them using discrete non-negative numbers: {0, 1, 2, 3, ....}. This type of variable receives the name of **count data**.

62

## Poisson distribution

- Imagine we are counting the number of red cars passing through an avenue per hour. We could use Poisson distribution to describe this data. Poisson distribution is generally used to describe the probability of a given number of events occurring on a fixed time/space interval. Thus, the Poisson distribution assumes that the events occur independently of each other and at a fixed interval of time and/or space. This discrete distribution is parametrized using only one value, (the rate, also commonly represented with the Greek letter $\lambda$). $\mu$ corresponds to the mean and also the variance of the distribution. The probability mass function of Poisson distribution is as follows:

$$f(x|\mu) = \frac{e^{-\mu}\mu^{x}}{x!}$$

63

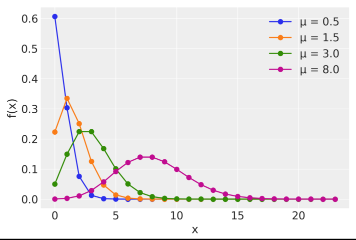
## Description of Equation

- $\mu$ is the average number of events per unit of time/space
- $x$ is a positive integer value {0, 1, 2, ...}
- $x!$ is the factorial of $\kappa$, that is, $\kappa! = \kappa\times(\kappa-1)\times(\kappa-2)\times\cdots\times2\times1$
- In the figure in the next slide, we can see some examples of the Poisson distribution family for different values of $\mu$:

```
mu_params = [0.5, 1.5, 3, 8]
x = np.arange(0, max(mu_params) * 3)
for mu in mu_params:
        y = stats.poisson(mu).pmf(x)
        plt.plot(x, y, 'o-', label=f'μ = {mu:3.1f}’)
plt.legend()
plt.xlabel('x’)
plt.ylabel('f(x)')
```

64

## Poisson Distribution Family (for Different $\mu$)



65

## Poisson Distribution is Discrete!

- Note that $\mu$ can be a float, but the output of the distribution is always an integer. In the figure in previous slide, the dots represent the values of the distribution, while the continuous lines are a visual aid to help us easily grasp the shape of the distribution. Remember, the Poisson distribution is a discrete distribution.
- The Poisson distribution can be seen as a special case of the binomial distribution when the number of trials $n$ is very large but the probability of success $p$ is very low. Without going into too much mathematical detail, let's try to clarify this statement. Following the car example, we can affirm that we either see the red car or we do not, thus we can use a binomial distribution. In that case we have:

$$x \sim Bin(n, p)$$

66

## Argument Why Poisson Distribution is Special Case of Binomial Distribution

- Then, the mean of the binomial distribution is:
$$E[x] = np$$
- And the variance is given by:
$$V[x] = np(1-p)$$
- But note that even if you are in a very busy avenue, the chance of seeing a red car compared to the total number of cars in a city is very small, and therefore we have:
$$n \gg p \rightarrow np \approx np(1-p)$$
- So, we can make the following approximation:
$$V[x] = np$$
- Now, the mean and the variance are represented by the same number, and we can confidently state that our variable is distributed as a Poisson distribution:
$$x \sim Poisson(\mu = np)$$

67

## Poisson Model Generates Fewer Zeros Compared to Data

- When counting things, one option is to not count a thing, that is, to get a zero. The number zero can generally occur for many reasons; we get a zero because we were counting red cars and a red car did not pass through the avenue or because we missed it (maybe we did not see that tiny red car behind that large truck). So if we use a Poisson distribution, we will notice, for example, when performing a posterior predictive check, that the model generates fewer zeros compared to the data. How do we fix that? We may try to address the exact cause of our model predicting fewer zeros than observed and include that factor in the model. But, as is often the case, it is enough and easier for our purposes just to assume that we have a mixture of two processes:
  - One modeled by a Poisson distribution with probability $\psi$
  - One giving extra zeros with probability $1-\psi$

68

## The Zero-Inflated Poisson Model

- This is known as *the zero-inflated Poisson (ZIP) model*. In some texts, you will find that $\psi$ represents the extra zeros and $1-\psi$ the probability of the Poisson. This is not a big deal; just pay attention to which is which for a concrete example.
- Basically, a ZIP distribution is:
$$p(y_j = 0) = 1 - \psi + (\psi)e^{-\mu}$$
$$p(y_j = k_i) = \psi \frac{\mu_i^x e^{-\mu}}{x_i!}$$
- Where $1-\psi$ is the probability of extra zeros.

69

## Example of ZIP Model

- To exemplify the use of the ZIP distribution, let's create a few synthetic data points:

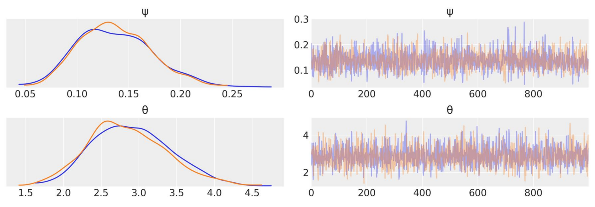
```
n = 100
θ_real = 2.5
ψ = 0.1
# Simulate some data
counts = np.array([(np.random.random() > (1-ψ)) * np.random.poisson(θ_real) for i in range(n)])
```

- We could easily implement the equations in the previous slide into a PyMC3 model. However, we can do something even easier: we can use the built-in ZIP distribution from PyMC3:

```
with pm.Model() as ZIP:
        ψ = pm.Beta('ψ', 1, 1)
        θ = pm.Gamma('θ', 2, 0.1)
        y = pm.ZeroInflatedPoisson('y', ψ, θ, observed=counts)
trace = pm.sample(1000)
```

70

## The Plots



71

## Poisson Regression and ZIP Regression

- The ZIP model may look a little dull, but sometimes we need to estimate simple distributions, such as this one, or others such as Poisson or Gaussian distributions. Anyway, we can also use Poisson or ZIP distributions as part of a linear model. As we saw with the logistic and softmax regressions, we can use an inverse link function to transform the result of a linear model into a variable in a range suitable to be used with other distributions than the normal. Following the same idea, we can now perform a regression analysis where the output variable is a count variable using a Poisson or a ZIP distribution. This time, we can use the exponential function, $e^{\cdot}$, as the inverse link function. This choice guarantees the values returned by the linear model are always positive:
$$\theta = e^{(\alpha + X\beta)}$$

72

## Dataset from Institute for Digital Research and Education

- To exemplify a ZIP-regression model implementation, we are going to work with a dataset taken from the *Institute for Digital Research and Education* (http://www.ats.ucla.edu/stat/data). We have 250 groups of visitors to a park. Here are some parts of the data per group:
  - The number of fish they caught (count)
  - How many children were in the group (child)
  - Whether they brought a camper to the park (camper)
- Using this data, we are going to build a model that predicts the number of caught fish as a function of the child and camper variables. We can use pandas to load the data:

```
fish_data = pd.read_csv('../data/fish.csv’)
```

  - You can explore the dataset using plots and/or a Pandas function, such as describe.

73

## ZIP-Regression Model

- For now, we are going to continue by implementing the ZIP_reg PyMC3 model:

```
with pm.Model() as ZIP_reg:
        ψ = pm.Beta('ψ', 1, 1)
        α = pm.Normal('α', 0, 10)
        β = pm.Normal('β', 0, 10, shape=2)
        θ = pm.math.exp(α + β[0] * fish_data['child'] + β[1] * fish_data['camper’])
        yl = pm.ZeroInflatedPoisson('yl', ψ, θ, observed=fish_data['count’])
trace_ZIP_reg = pm.sample(1000)
```

- Camper is a binary variable with a value of 0 for not-camper and 1 for camper. A variable indicating the absence/presence of an attribute is usually denoted as a dummy variable or indicator variable. Note that when camper takes the value of 0, the term involving $\beta_1$ will also be 0 and the model reduces to a regression with a single independent variable.

74

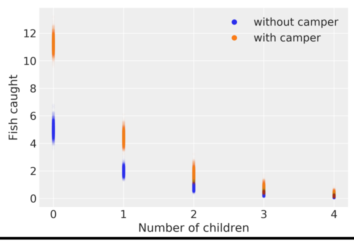
## Generating Figure

- To better understand the results of our inference, let's do a plot:

```
children = [0, 1, 2, 3, 4]
fish_count_pred_0 = []
fish_count_pred_1 = []
for n in children:
        without_camper = trace_ZIP_reg['α'] + trace_ZIP_reg['β'][:,0] * n
        with_camper = without_camper + trace_ZIP_reg['β'][:,1]
        fish_count_pred_0.append(np.exp(without_camper))
        fish_count_pred_1.append(np.exp(with_camper))
plt.plot(children, fish_count_pred_0, 'C0.', alpha=0.01)
plt.plot(children, fish_count_pred_1, 'C1.', alpha=0.01)
plt.xticks(children);
plt.xlabel('Number of children’)
plt.ylabel('Fish caught’)
plt.plot([], 'C0o', label='without camper’)
plt.plot([], 'C1o', label='with camper’)
plt.legend()
```

75

## The Plot



76

## Plot Explanation

- From the figure in the previous slide, we can see that the higher the number of children, the lower the number of fish caught. Also people that travel with a camper generally catch more fish. If you check the $\beta$ coefficients for child and camper, you will see that we can say the following:
  - For each additional child, the expected count of the fish caught decreases by $\approx 0.4$
  - Being in a camper increases the expected count of the fish caught by $\approx 2$
- We arrive at these values by taking the exponential of the $\beta_1$ and $\beta_2$ coefficients, respectively.

77

## Robust Logistic Regression

- We just saw how to fix an excess of zeros without directly modeling the factor that generates them. A similar approach, suggested by Kruschke, can be used to perform a more robust version of logistic regression. Remember that in logistic regression, we model the data as binomial, that is, zeros and ones. So it may happen that we find a dataset with unusual zeros and/or ones. Take, as an example, the iris dataset that we already saw, but with some added *intruders*:

```
iris = sns.load_dataset("iris")
df = iris.query("species == ('setosa', 'versicolor')")
y_0 = pd.Categorical(df['species']).codes
x_n = 'sepal_length’
x_0 = df[x_n].values
y_0 = np.concatenate((y_0, np.ones(6, dtype=int)))
x_0 = np.concatenate((x_0, [4.2, 4.5, 4.0, 4.3, 4.2, 4.4]))
x_c = x_0 - x_0.mean()
plt.plot(x_c, y_0, 'o', color='k');
```

78

## Mixture Model

- Here, we have some versicolors (1s) with an unusually short sepal length. We can fix this with a mixture model. We are going to say that the output variable comes with the $\pi$ probability by random guessing, or with the $1-\pi$ probability from a logistic regression model. Mathematically we have:

$$p = \pi 0.5 + (1-\pi)\text{logistic}(\alpha + X\beta)$$

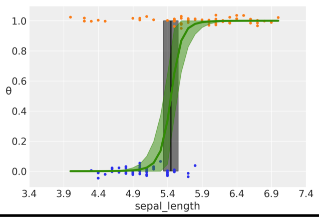
- Notice that when $\pi = 1$, we get $p = 0.5$, and for $\pi = 0$, we recover the expression for logistic regression. Implementing this model is a straightforward modification of the first model from this chapter:

```
with pm.Model() as model_rlg:
        α = pm.Normal('α', mu=0, sd=10)
        β = pm.Normal('β', mu=0, sd=10)
        μ = α + x_c * β
        θ = pm.Deterministic('θ', pm.math.sigmoid(μ))
        bd = pm.Deterministic('bd', -α/β)
        π = pm.Beta('π', 1., 1.)
        p = π * 0.5 + (1 - π) * θ
        yl = pm.Bernoulli('yl', p=p, observed=y_0)
trace_rlg = pm.sample(1000)
```

79

## The Results

- If we compare these results with those from model_0 (the first model in this chapter), we will see that we get more or less the same boundary. As we can see by comparing the figure on the right with the figure in slide #18.
- You may also want to compute the summary for model_0 and model_rlg to compare the values of the boundary according to each model.



80

## The GLM Module

- As we discussed at the beginning of this chapter, linear models are very useful statistical tools. Extensions such as the ones we we saw in this chapter make them even more general tools. For that reason, PyMC3 includes a module to simplify the creation of linear models: the **Generalized Liner Model** (**GLM**) module. For example, a simple linear regression will be as follows:

```
with pm.Model() as model:
        glm.glm('y ~ x', data)
trace = sample(2000)
```

81

## Customize GLM Module

- The second line of the preceding code takes care of adding priors for the intercept and for the slope. By default, the intercept is assigned a flat prior, and the slopes $N(0,1\times10^6)$ an prior. Note that the **maximum a posteriori** (**MAP**) of the default model will be essentially equivalent to the one obtained using the ordinary least squared method. These is totally fine as a default linear regression; you can change it using the priors argument. The GLM module also adds a Gaussian likelihood by default. You can change it using the family argument; you can choose from the following options: Normal (default), StudentT, Binomial, Poisson, or NegativeBinomial.
- To describe a statistical model, the GLM module uses Patsy (https://patsy.readthedocs. io/en/latest/index.html), which is a Python library that provides a *formula mini-language* syntax inspired by the one used in R and S. In the previous code block, $y \sim x$ specifies we have an output variable, $y$, that we want to estimate as a linear function of $x$.

82