Data Science for Biodiversity Scientists

An introduction to concepts and practices

Timothée Poisot

2023-08-21

Table of contents

Preface

Data science is now an established methodology to study biodiversity, and this is a problem.

This may be an opportunity when it comes to advancing our knowledge of biodiversity, and in particular when it comes to translating this knowledge into action (Tuia *et al.* 2022); but make no mistake, this is a problem for us, biodiversity scientists, as we suddenly need to develop competences in an entirely new field. And as luck would have it, there are easier fields to master than data science. The point of this book, therefore, is to provide an introduction to fundamental concepts in data science, from the perspective of a biodiversity scientist, by using examples corresponding to real-world use-cases of these techniques.

But what do we mean by *data science*? Most science, after all, relies on data in some capacity. What falls under the umbrella of data science is, in short, embracing in equal measure quantitative skills (mathematics, machine learning, statistics), programming, and domain expertise, in order to solve well-defined problems. A core tenet of data science is that, when using it, we seek to "deliver actionable insights", which is MBA-speak for "figuring out what to do next". One of the ways in which this occurs is by letting the data speak, after they have been, of course, properly cleaned and transformed and engineered beyond recognition. This entire process is driven by (or subject to, even) domain knowledge. There is no such thing as data science, at least not in a vacuum: there is data science as a methodology applied to a specific domain.

Before we embark into a journey of discovery on the applications of data science to biodiversity, allow me to let you in on a little secret: data *science* is a little bit of a misnomer.

To understand why, it helps to think of science (the application of the scientific method, that is) as cooking. There are general techniques one must master, and specific steps and cultural specifics, and there is a final product. When writing this preface, I turned to my shelf of cookbooks, and picked my two favorites: Robuchon's *The Complete Robuchon* (a no-nonsense list of hundreds of recipes with no place for improvisation), and Bianco's *Pizza, Pasta, and Other Food I Like* (a short volume with very few pizza and pasta, and wonderful discussions about the importance of humility, creativity, and generosity). Data science, if

it were cooking, would feel a lot like the second. Deviation from the rules (they are mostly recommendations, in fact) is often justifiable if you feel like it. But this improvisation requires good skills, a clear mental map of the problem, and a library of patterns that you can draw from.

This book will not get you here. But it will speed up the process, by framing the practice of data science as a natural way to conduct research on biodiversity.

Chapter 1

Introduction

This book started as a collection of notes from several classes I gave in the Department of Biological Sciences at the Université de Montréal, as well as a few workshops I ran for the Québec Centre for Biodiversity Sciences. In teaching data synthesis, data science, and machine learning to biology students, I realized that the field was missing a stepping stone to proficiency. There are excellent manuals covering the mathematics of data science and machine learning **REFS**; there are many good papers giving overviews of some applications of data science to biological problems **REFS**; and there are, of course, thousands of tutorials about how to write code. But one thing that students commonly called for was an attempt to tie concepts together. This is this attempt.

1.1 Core concepts in data science

1.1.1 EDA

1.1.2 Clustering and regression

1.1.3 Supervised and unsupervised

1.1.4 Training, testing, and validation

1.1.5 Transformations and feature engineering

1.2 An overview of the content

1.3 How to read this book

order of the chapters is important (one thing at a time, chapters follow one another)

note on the meaning of colors

1.4 Some rules about this book

When I started aggregating these notes, I decided on a series of four rules. No code, no simulated data, no long list of model, and no `iris`. In this section, I will go through *why* I decided to adopt these rules, and how it should change the way you interact with the book.

1.4.1 No code

This is, maybe, the most surprising rule, because data science *is* programming (in a sense). But sometimes there is so much focus on programming that we lose track of the other, important aspects of the practice of data science: abstractions, relationship with data, and domain knowledge.

This book *did* involve a lot of code. Specifically, this book was written using *Julia* (Bezanson *et al.* 2017), and every figure is generated by a notebook, and they are part of the material I use when teaching from this content in the classroom. But code is *not* a universal language, and unless you are really familiar with the language, code can obfuscate. I had no intention to write a *Julia* book (or an *R* book, or a *Python* book). The point is to think about data science applied to ecological research, and I felt like it would be more inclusive to do this in a language agnostic way.

And finally, code rots. Code with more dependencies rots faster. It take a single change in the API of a package to break the examples, and then you are left with a very expensive monitor stand. With a few exceptions, the examples in this book do not use complicated packages either.

1.4.2 No simulated data

1.4.3 No model zoo

My favorite machine learning package is *MLJ* (Blaom *et al.* 2020). When given a table of labels and a table of features, it will give back a series of models that match with these data. It speeds up the discovery of models considerably, and is generally a lot more informative than trying to read from a list of possible techniques. If I have questions about an algorithm from this list, I can start reading more documentation about how it works.

Reading a long enumeration of things is boring; unless it's sung by Yakko Warner, I'm not interested, and I refuse to inflict it on people. But more importantly, these enumerations of models often distract from thinking about the problem we want to solve in more abstract terms. I rarely wake up in the morning and think "oh boy I can't wait to train a SVM today"; chances are, my thought process will be closer to "I need to tell the mushroom people where

I think the next good foraging locations will be". The rest, is implementation details.

In fact, 90% of this book uses only two models: linear regression, and the Naïve Bayes Classifier. Some other models are involved in a few chapters, but these two models are breathtakingly simple, work surprisingly well, run fast, and can be tweaked to allow us to build deep intuitions about how machines learn. They are perfect for the classroom, and give us the freedom to spent most of our time thinking about how we interact with models, and why, and how we make methodological decisions.

1.4.4 No `iris`

From a teaching point of view, the `iris` dataset is like hearing Smash Mouth in a movie trailer, in that it tells you two things with absolute certainty. First, that you are indeed watching a movie trailer. Second, that you could be watching Shrek instead.

But there is a far more important reason not to use `iris`.

Thankfully, there are alternatives! The most broadly known is `penguins`, which was collected by ecologists (Gorman *et al.* 2014), and is generally a robust . We won't use `penguins` either. It's a fine dataset, but at this point there is little that we can write around it that would be new, or exciting, or relevant outside of the cottage industry of people who write about machine learning on *Medium*.

Chapter 2

Clustering

As we mentioned in the introduction, a core idea of data science is that things that look the same (in that, when described with data, they resemble one another) are likely to be the same. Although this sounds like a simplifying assumption, this can provide the basis for a very powerful technique in which we *create* groups in data that have no labels. This task is called unsupervised clustering: we seek to add a *label* to each observation, in order to form groups, and the data we work from do *not* have a label that we can use to train a model.

2.1 A digression: which birds are red?

Before diving in, it is a good idea to ponder a simple case. We can divide everything in just two categories: things with red feathers, and things without red feathers. An example of a thing with red feathers is the Northern Cardinal (*Cardinalis cardinalis*), and things without red feathers are the iMac G3, Haydn's string quartets, and of course the Northern Cardinal (*Cardinalis cardinalis*).

See, biodiversity data science is complicated, because it tends to rely on the assumption that we can categorize the natural world, and the natural world (mostly in response to natural selection) comes up with ways to be, well, diverse. In the Northern Cardinal, this is shown in males having red feathers, and females having mostly brown feathers. Before moving forward, we need to consider ways to solve this issue, as this issue will come up *all the time.*

The first mistake we have made is that the scope of objects we want to classify, which we will describe as the "domain" of our classification, is much too broad: there are few legitimate applications where we will have a dataset with Northern Cardinals, iMac G3s, and Haydn's string quartets. Picking a reasonable universe of classes would have solved our problem a little. For example, among the things that do not have red feathers are the Mourning Dove, the

Kentucky Warbler, and the House Sparrow.

The second mistake that we have made is improperly defining our classes; bird species exhibit sexual dimorphism (not in an interesting way, like wrasses, but you let's still give them some credit for trying). Assuming that there is such a thing as a Northern Cardinal is not necessarily a reasonable assumption! And yet, the assumption that a single label is a valid representation of non-monomorphic populations is a surprisingly common one, with actual consequences for the performance of image classification algorithms (Luccioni & Rolnick 2023). This assumption reveals a lot about our biases: male specimens are over-represented in museum collections, for example (Cooper *et al.* 2019). In a lot of species, we would need to split the taxonomic unit into multiple groups in order to adequately describe them.

The third mistake we have made is using predictors that are too vague. The "presence of red feathers" is not a predictor that can easily discriminate between the Northen Cardinal (yes for males, sometimes for females), the House Finch (a little for males, no for females), and the Red-Winged Black Bird (a little for males, no for females). In fact, it cannot really capture the difference between red feathers for the male House Finch (head and breast) and the male Red Winged Black Bird (wings, as the name suggests).

The final mistake we have made is in assuming that "red" is relevant as a predictor. In a wonderful paper, Cooney *et al.* (2022) have converted the color of birds into a bird-relevant colorimetric space, revealing a clear latitudinal trend in the ways bird colors, as perceived by other birds, are distributed. This analysis, incidentally, splits all species into males and females. The use of a color space that accounts for the way colors are perceived is a fantastic example of why data science puts domain knowledge front and center.

Deciding which variables are going to be accounted for, how the labels will be defined, and what is considered to be within or outside the scope of the classification problem is *difficult*. It requires domain knowledge (you must know a few things about birds in order to establish criteria to classify birds), and knowledge of how the classification methods operate (in order to have just the right amount of overlap between features in order to provide meaningful estimates of distance).

2.2 The problem: classifying pixels from an image

Throughout this chapter, we will work on a single image – we may initially balk at the idea that an image is data, but it is! Specifically, an image is a series of instances (the pixels), each described by their position in a multidimensional colorimetric space. Greyscale images have one dimension, and images in color will have three: their red, green, and blue channels. Not only are images data, this specific dataset is going to be far larger than many of the datasets we will

work on in practice: the number of pixels we work with is given by the product of the width and height of the image!

In fact, we are going to use an image with a lot more dimensions: the data in this chapter are coming from a Landsat 9 image Vermote *et al.* (2016), for which we have access to 7 different bands (the full data product has more bands, but we will not use them all.

Band number	Information
1	Aerosol
2	Visible blue
3	Visible red
4	Visible green
5	Near-infrared (NIR)
6	Short wavelength IR (SWIR 1)
7	SWIR 2

From these channels, we can reconstruct an approximation of what the landscape looked like (by using the red, green, and blue channels) – this information is presented in Figure 2.1 . Or is it? If we were to invent a time machine, and go stand directly under Landsat 9 at the exact center of this scene, and look around, what would we see? We would see colors, and they would admit a representation as a three-dimensional vector of red, green, and blue. But we would see so much more than that! And even if we were to stand within a pixel, we would see a *lot* of colors. And texture. And depth. We would see something entirely different from this map; and we would be able to draw a lot more inferences about our surroundings than what is possible by knowing the average color of a 30x30 meters pixel.

But just like we can get more information that Landsat 9, so to can Landsat 9 out-sense us when it comes to getting information. In the same way that we can extract a natural color composite out of the different channels, we can extract a fake color one to highlight differences in the landscape; in Figure 2.2, we show such a fake color composite, that is particularly efficient at drawing our attention to the location of water in this area.

Both Figure 2.2 and Figure 2.1 represent the same physical place at the same moment in time; but through them, we are looking at this place with very different purposes. This is not an idle observation, but a core notion in data science: what we measure defines what we can see. In order to tell something meaningful about this place, we need to look at it in the "right" way.

So far, we have looked at this area by combining the raw data. Depending on the question we have in mind, they may not be the *right* data. In fact, they may not hold information that is relevant to our question *at all*; or worse, they can hold more noise than signal. Looking at Figure 2.1, we might wonder, "where

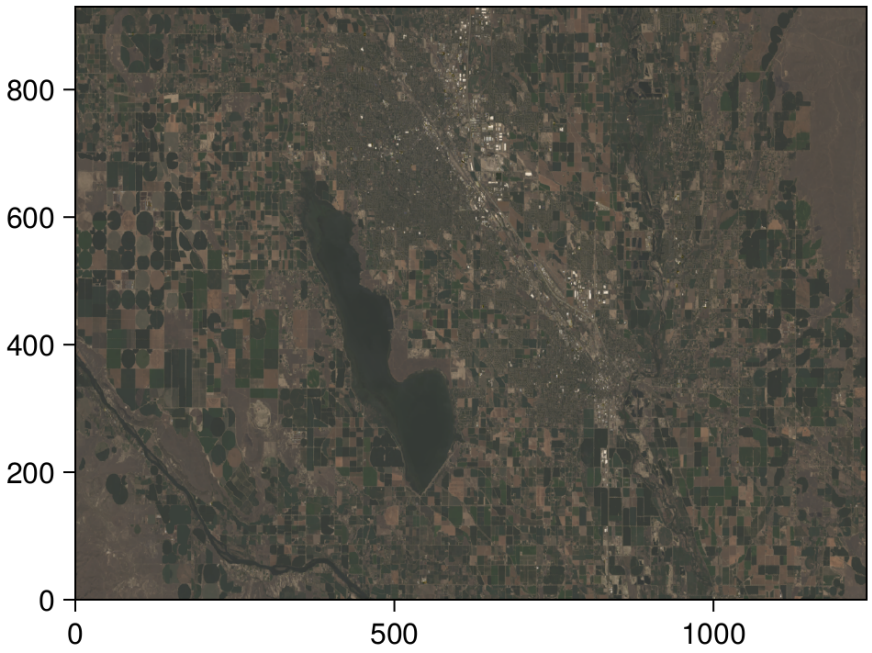

Figure 2.1: The Landsat 9 data are combined into the "Natural Color" image, in which the red, green, and blue bands are mapped to their respective channels. This looks eerily like the way we perceive the landscape. Note how little difference there is between the large body of water and the surrounding crops. This emphasizes that the combination of channels we use will limit our ability to separate the pixels into groups.

are the fields?". And based on our knowledge of what plants do, we can start thinking about this question in a different way. Specifically, "is there a series of features of fields that are not shared by non-fields?". But this a complicated question to answer, and so we can simplify this by asking, "how can I combine data from the image to know if there is a plant?".

One way to do this is to calculate the normalized difference vegetation index, or NDVI (Kennedy & Burbach 2020). NDVI is derived from the band data (we will see how in a minute), and is an adequate heuristic to make a difference between vegetation, barren soil, and water. Because we are specifically thinking about fields, we can also consider the NDWI (water) and NDMI (moisture) dimensions: taken together, these information will represent every pixel in a three-dimensional space, telling us whether there are plants (NDVI), whether they are stressed (NDMI), and whether this pixel is a water body (NDWI).

Because there are a few guidelines (educated guesses, in truth, and the jury is

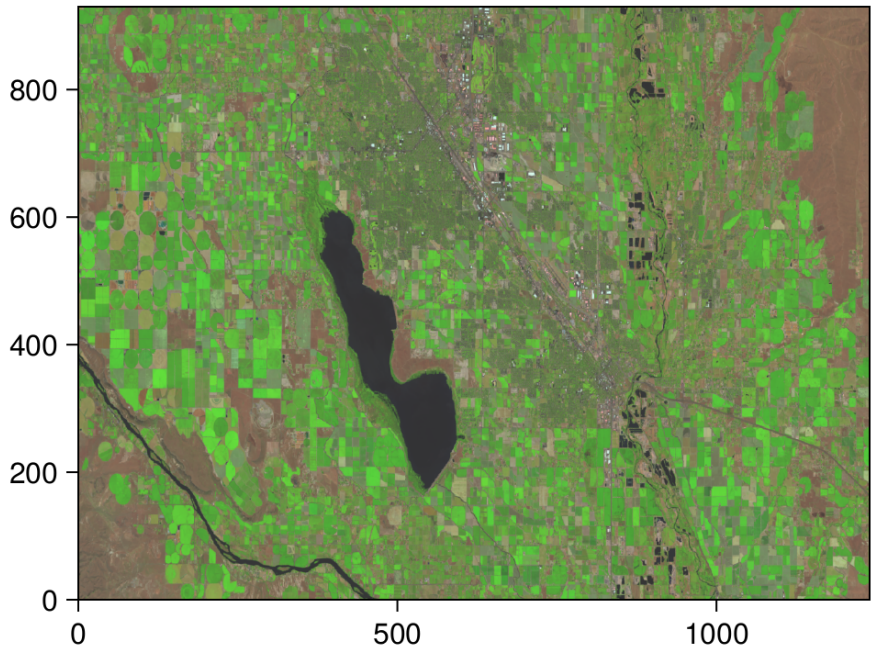

Figure 2.2: The same landscape as above is now presented in a fake color composite, where SWIR is mapped to the red channel, NIR to the green channel, and red to the blue channel. This highlights different values in the landscape, but is no more or less "real" than the true color composite. It is a visualization of the data, that represents different choices, questions, and assumptions. Compared to the Natural Color composite, this version of the data highlights the water, areas with vegetation, and more arid areas.

still out on the "educated" part) about the values, we can look at the relationship between the NDVI and NDMI data Figure 2.3. For example, NDMI values around -0.1 (note how there is a strong cluster of points here) are low-canopy cover with low water stress; NDVI values from 0.2 to 0.5 are good candidates for moderately dense crops.

By picking these three values, instead of simply looking at the clustering of all the bands in the raw data, we are starting to refine what the algorithm see, through the lens of what we know is important about the system.

2.3 The theory behind *k*-means clustering

In order to understand the theory underlying *k*-means, we will work backwards from its output. As a method for unsupervised clustering, *k*-means will return a vector of *class memberships*, which is to say, a list that maps each observation

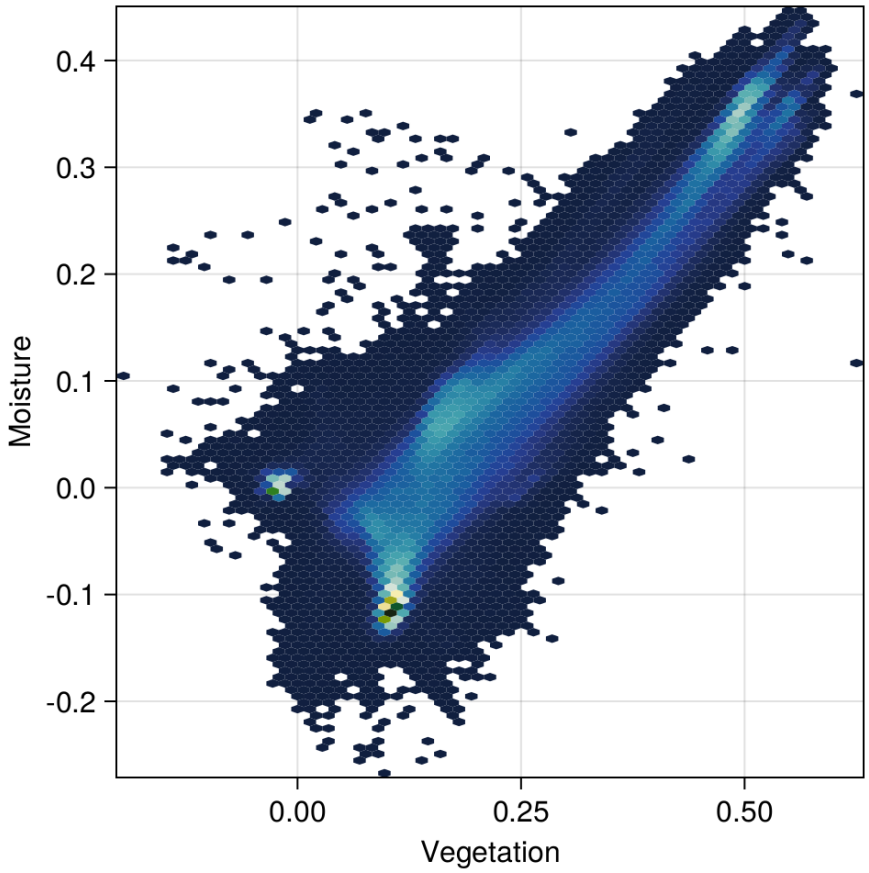

Figure 2.3: The pixels acquired from Landsat 8 exist in a space with many different dimensions (one for each band). Because we are interested in a landscape classification based on water/vegetation data, we use the NDVI, NDMI, and NDWI combinations of bands. These are *derived* data, and represent an instance of feature engineering: we have derived these values from the raw data.

(pixel, in our case) to a class (tentatively, a cohesive landscape unit). What this means is that *k*-means is a transformation, taking as its input a vector with three dimensions (red, green, blue), and returning a scalar (an integer, even!), giving the class to which this pixel belongs. These are the input and output of our blackbox, and now we can start figuring out its internals.

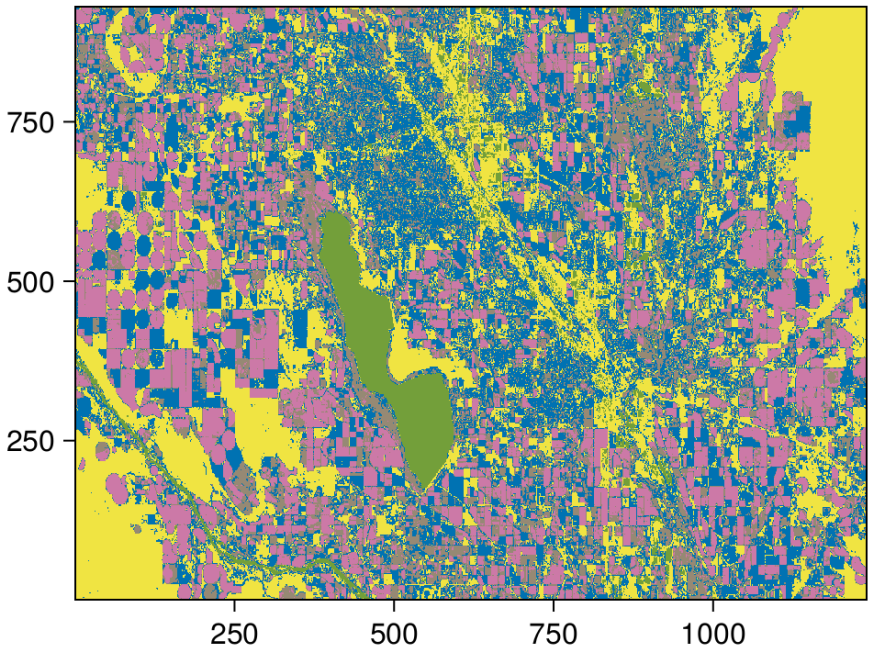

Figure 2.4: After iterating the k-means algorithm, we obtain a classification for every pixel in the landscape. This classification is based on the values of NDVI, NDMI, and NDWI indices, and therefore groups pixels based on a specific hypothesis. This clustering was produced using $k = 5$, *i.e.* we want to see what the landscape would look like when divided into five categories.

2.3.1 Overview of the algorithms

2.4 Identification of the optimal number of clusters

2.5 Application: optimal clustering of the satellite image data

2.6 Alternatives and improvements

EM

k-median

k-medoids

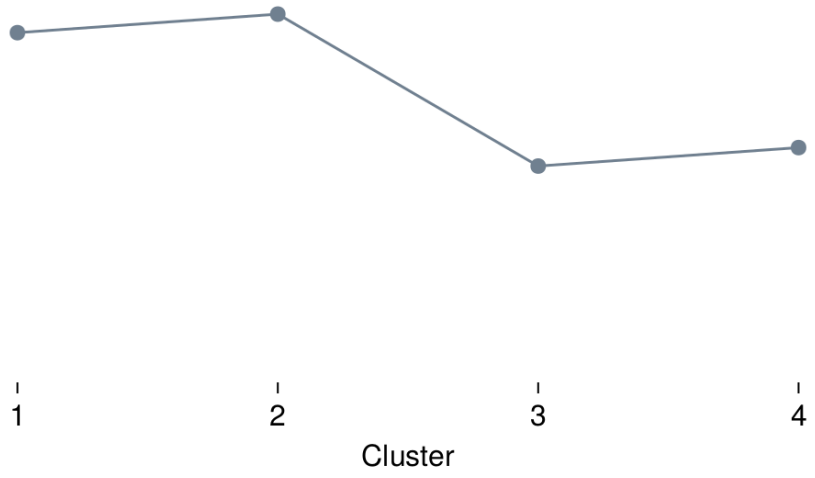

Figure 2.5: Number of pixels assigned to each class in the final landscape classification. In most cases, *k*-means will create clusters with the same number of points in them. This may be an issue, or this may be a way to ensure that whatever classes are produced will be balanced in terms of their representation.

Chapter 3

Gradient descent

As we progress into this book, the process of delivering a trained model is going to become increasingly complex. In Chapter 2, we work with a model that did not really require training (but did require to pick the best hyper-parameter). In this chapter, we will only increase complexity very slightly, by considering how we can train a model when we have a reference dataset to compare to.

Doing do will require to introduce several new concepts, and so the "correct" way to read this chapter is to focus on the high-level process. The problem we will try to solve (which is introduced in Section 3.2) is very simple; in fact, the empirical data looks more fake than many simulated datasets!

3.1 A digression: what is a trained model?

3.2 The problem: how many interactions in a food web?

One of the earliest observation that ecologists made about food webs is that when there are more species, there are more interactions. A remarkably insightful crowd, food web ecologists. Nevertheless, it turns out that this apparently simple question had received a few different answers over the years. In

The initial model was proposed by Cohen & Briand (1984): the number of interactions L scales linearly with the number of species S. After all, we can assume that when averaging over many consumers, there will be an average diversity of resources they consume, and so the number of interactions could be expressed as $L \approx b \times S$.

Not so fast, said Martinez (1992). When we start looking a food webs with more species, the increase of L with regards to S is superlinear. Thinking in ecologi-

cal terms, maybe we can argue that consumers are flexible, and that instead of sampling a set number of resources, they will sample a set proportion of the number of consumer-resource combinations (of which there are S^2). In this interpretation, $L \approx b \times S^2$.

But the square term can be relaxed; and there is no reason not to assume a power law, with $L \approx b \times S^a$. This last formulation has long been accepted as the most workable one, because it is possible to approximate values of its parameters using other ecological processes (Brose *et al.* 2004).

The "reality" (*i.e.* the relationship between S and L that correctly accounts for ecological constraints, and fit the data as closely as possible) is a little bit different than this formula (MacDonald *et al.* 2020). But for the purpose of this chapter, figuring out the values of a and b from empirical data is a very instructive exercise.

In Figure 3.1, we can check that there is a linear relationship between the natural log of the number of species and the natural log of the number of links. This is not surprising! If we assume that $L \approx b \times S^a$, then we can take the log of both sides, and we get $\log L \approx a \times \log S + \log b$. This is linear model, and so we can estimate its parameters using linear regression!

3.3 Gradient descent

Gradient descent is built around a remarkably simple intuition: knowing the formula that gives rise to our prediction, and the value of the error we made for each point, we can take the derivative of the error with regards to each parameter, and this tells us how much this parameter contributed to the error. Because we are taking the derivative, we can futher know whether to increase, or decrease, the value of the parameter in order to make a smaller error next time.

In this section, we will use linear regression as an example, because it is the model we have decided to use when exploring our ecological problem in Section 3.2, and because it is suitably simple to keep track of everything when writing down the gradient by hand.

Before we start assembling the different pieces, we need to decide what our model is. We have settled on a linear model, which will have the form $\hat{y} = m \times x + b$. The little hat on $\hat{y}$ indicates that this is a prediction. The input of this model is x, and its parameters are m (the slope) and b (the intercept). Using the notation we adopted in Section 3.2, this would be $\hat{l} = a \times s + b$, with $l = \log L$ and $s = \log S$.

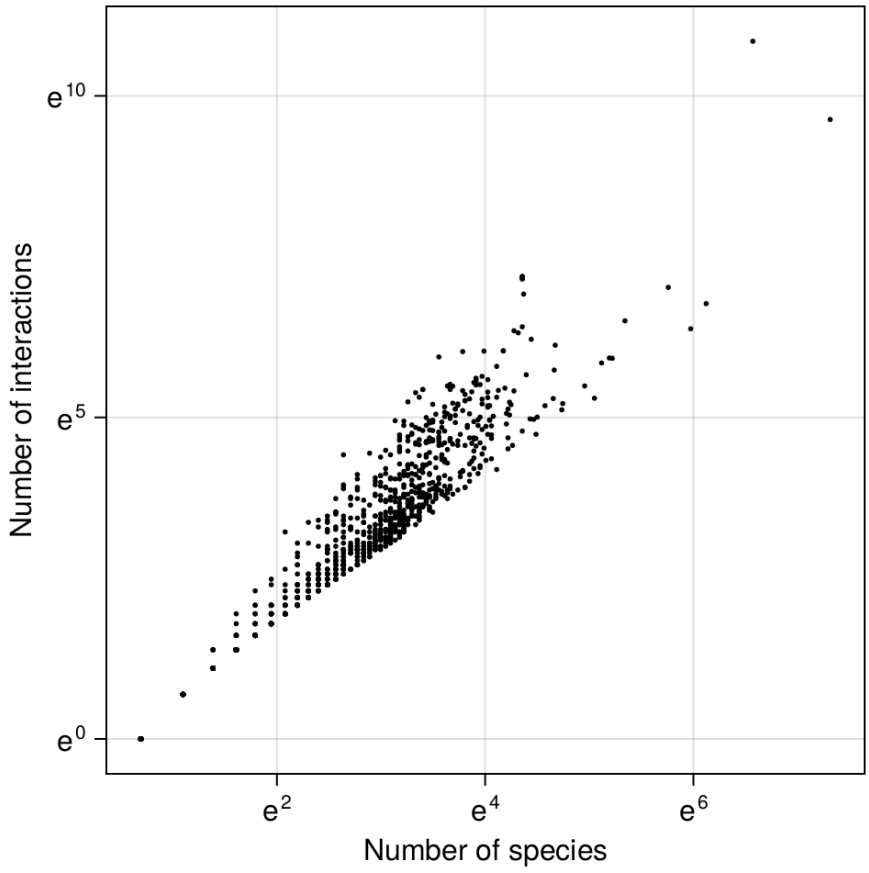

Figure 3.1: We have assumed that the relationship between L and S could be represented by $L \approx b \times S^a$, which gave us a reason to take the natural log of both variables. On this figure, we see that the relationship between the logs look linear, which means that linear regression has a good chance of estimating the values of the parameters.

3.3.1 Defining the loss function

The loss function is an important concept for anyone attempting to compare predictions to outcomes: it quantifies how far away an ensemble of predictions is from a benchmark of known cases. There are many loss functions we can use, and we will indeed use a few different ones in this book. But for now, we will start with a very general understanding of what these functions *do*.

Think of prediction as throwing a series of ten darts on ten different boards. In this case, we know what the correct outcome is (the center of the board, I assume, although I can be mistaken since I have only played darts once, and lost). A cost function would be any mathematical function that compares the position of each dart on each board, the position of the correct event, and returns a score that informs us about how poorly our prediction lines up with the reality.

In the above example, you may be tempted to say that we can take the Euclidean distance of each dart to the center of each board, in order to know, for each point,

how far away we landed. Because there are several boards, and because we may want to vary the number of boards while still retaining the ability to compare our performances, we would then take the average of these measures.

We will note the position of our dart as being $\hat{y}$, the position of the center as being y (we will call this the *ground truth*), and the number of attempts n, and so we can write our loss function as

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{3.1}$$

In data science, things often have multiple names. This is true of loss functions, and this will be even more true on other things later.

This loss function is usually called the MSE (Mean Standard Error), or L2 loss, or the quadratic loss, because the paths to machine learning terminology are many. This is a good example of a loss function for regression (and we will discuss loss functions for classification later in this book). There are alternative loss functions to use for regression problems in Table 3.1.

Table 3.1: List of common los
tions for regression problems

Measure	Expression	Remarks
Mean Squared Error (MSE, L2)	$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$	Large errors are (proportionally) more penalized because of the squaring
Mean Absolute Error (MAE, L1)	$\frac{1}{n}\sum_{i=1}^{n}\|y_i - \hat{y}_i\|$	Error measured in the units of the response variable
Root Mean Square Error (RMSE)	$\sqrt{\text{MSE}}$	Error measured in the units of the response variable
Mean Bias Error	$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)$	Errors *can* cancel out, but this can be used as a measure of positive/negative bias

Throughout this chapter, we will use the L2 loss (Equation 3.1), because it has *really* nice properties when it comes to taking derivatives, which we will do a lot of. In the case of a linear model, we can rewrite Equation 3.1 as

$$f = \frac{1}{n}\sum(y_i - m \times x_i - b)^2 \tag{3.2}$$

There is an important change in Equation 3.2: we have replaced the prediction $\hat{y}_i$ with a term that is a function of the predictor x_i and the model parameters: this means that we can calculate the value of the loss as a function of a pair of values (x_i, y_i), and the model parameters.

3.3.2 Calculating the gradient

With the loss function corresponding to our problem in hands (Equation 3.2), we can calculate the gradient. Given a function that is scalar-valued (it returns a single value), taking several variables, that is differentiable, the gradient of this function is a vector-valued (it returns a vector) function; when evaluated at a specific point, this vectors indicates both the direction and the rate of fastest increase, which is to say the direction in which the function increases away from the point, and how fast it moves.

We can re-state this definition using the terms of the problem we want to solve. At a point $p = [m \quad b]^\top$, the gradient ∇f of f is given by:

$$\nabla f\,(p) = \begin{bmatrix} \frac{\partial f}{\partial m}(p) \\ \frac{\partial f}{\partial b}(p) \end{bmatrix}. \tag{3.3}$$

This indicates how changes in m and b will *increase* the error. In order to have a more explicit formulation, all we have to do is figure out an expression for both of the partial derivatives. In practice, we can let auto-differentiation software calculate the gradient for us (Innes 2018); these packages are now advanced enough that they can take the gradient of code directly.

Solving $(\partial f/\partial m)(p)$ and $(\partial f/\partial c)(p)$ is easy enough:

$$\nabla f\,(p) = \begin{bmatrix} -\frac{2}{n}\sum\left[x_i \times (y_i - m \times x_i - b)\right] \\ -\frac{2}{n}\sum\left(y_i - m \times x_i - b\right) \end{bmatrix}. \tag{3.4}$$

Note that both of these partial derivatives have a term in $2n^{-1}$. Getting rid of the 2 in front is very straightforward! We can modify Equation 3.2 to divide by $2n$ instead of n. This modified loss function retains the important characteristics: it increases when the prediction gets worse, and it allows comparing the loss with different numbers of points. As with many steps in the model training process, it is important to think about *why* we are doing certain things, as this can enable us to make some slight changes to facilitate the analysis.

With the gradient written down in Equation 3.4, we can now think about what it means to *descend* the gradient.

3.3.3 Descending the gradient

Recall from Section 3.3.2 that the gradient measures how far we *increase* the function of which we are taking the gradient. Therefore, it measures how much each parameter contributes to the loss value. Our working definition for a trained model is "one that has little loss", and so in an ideal world, we could find a point p for which the gradient is as small as feasible.

Because the gradient measures how far away we increase error, and intuitive way to use it is to take steps in the *opposite* direction. In other words, we can update the value of our parameters using $p := p - \nabla f(p)$, meaning that we subtract from the parameter values their contribution to the overall error in the predictions.

But, as we will discuss further in Section 3.3.4, there is such a thing as "too much learning". For this reason, we will usually not move the entire way, and introduce a term to regulate how much of the way we actually want to descend the gradient. Our actual scheme to update the parameters is

$$p := p - \eta \times \nabla f(p) \,. \tag{3.5}$$

This formula can be *iterated*: with each successive iteration, it will get us closer to the optimal value of p, which is to say the combination of m and b that minimizes the loss.

3.3.4 A note on the learning rate

The error we can make on the first iteration will depend on the value of our initial pick of parameters. If we are *way off*, especially if we did not re-scale our predictors and responses, this error can get very large. And if we make a very large error, we will have a very large gradient, and we will end up making very big steps when we update the parameter values. There is a real risk to end up over-compensating, and correcting the parameters too much.

In order to protect against this, in reality, we update the gradient only a little, where the value of "a little" is determined by an hyper-parameter called the *learning rate*, which we noted η. This value will be very small (much less than one). Picking the correct learning rate is not simply a way to ensure that we get correct results (though that is always a nice bonus), but can be a way to ensure that we get results *at all*. The representation of numbers in a computer's memory is tricky, and it is possible to create an overflow: a number so large it does not fit within 64 (or 32, or 16, or however many we are using) bits of memory.

The conservative solution of using the smallest possible learning rate is not really effective, either. If we almost do not update our parameters at every epoch, then we will take almost forever to converge on the correct parameters. Figuring out the learning rate is an example of hyper-parameter tuning, which we will get back to later in this book.

3.4 Application: how many links are in a food web?

We will not get back to the problem exposed in Figure 3.1, and use gradient descent to fit the parameters of the model defined as $\hat{y} \approx \beta_0 + \beta_1 \times x$, where,

using the notation introduced in Section 3.2, $\hat{y}$ is the natural log of the number of interactions (what we want to predict), x is the natural log of the species richness (our predictor), and β_0 and β_1 are the parameters of the model.

3.4.1 The things we won't do

At this point, we could decide that it is a good idea to transform our predictor and our response, for example using the z-score. But this is not really required here; we know that our model will give results that make sense in the units of species and interactions (after dealing with the natural log, of course). In addition, as we will see in Chapter 5, applying a transformation to the data too soon can be a dangerous thing. We will have to live with raw features for a few more chapters.

In order to get a sense of the performance of our model, we will remove some of the data, meaning that the model will not learn on these data points. We will get back to this practice (cross-validation) in a lot more details in Chapter 4, but for now it is enough to say that we hide 20% of the dataset, and we will use them to evaluate how good the model is as it trains. The point of this chapter is not to think too deeply about cross-validation, but simply to develop intuitions about the way a machine learns.

3.4.2 Starting the learning process

In order to start the gradient descent process, we need to decide on an initial value of the parameters. There are many ways to do it. We could work our way from our knowledge of the system; for example $b < 1$ and $a = 2$ would fit relatively well with early results in the food web literature. Or we could draw a pair of values (a, b) at random. Looking at **?@fig-gradientdescent-data**, it is clear that our problem is remarkably simple, and so presumably either solution would work.

3.4.3 Stopping the learning process

The gradient descent algorithm is entirely contained in **TODO EQ**, and so we only need to iterate several times to optimize the parameters. How long we need to run the algorithm for depends on a variety of factors, including our learning rate (slow learning requires more time!), our constraints in terms of computing time, but also how good we need to model to be.

One usual approach is to decide on a number of iterations, and to check how rapidly the model seems to settle on a series of parameters. But more than this, we also need to ensure that our model is not learning *too much* from the data. This would result in over-fitting, in which the models gets better on the data we used to train it, and worse on the data we kept hidden from the training! In Table 3.2, we present the RMSE loss for the training and testing datasets, as well as the current estimates of the values of the parameters of the linear model.

The number of iterations over which we train the model is called the number of epochs, and is an hyper-parameter of the model.

Step	Loss (training)	Loss (testing)	β□	β□
1	3.81952	3.59185	0.4	0.2
10	2.93171	2.73714	0.485703	0.226401
30	1.69896	1.55981	0.63612	0.271203
100	0.525114	0.474702	0.901006	0.337528
300	0.379423	0.365137	1.03407	0.312931
1000	0.315998	0.305429	1.10702	0.115729
3000	0.218368	0.208811	1.25051	-0.297807
10000	0.160111	0.143257	1.43959	-0.84272
20000	0.15805	0.138455	1.47878	-0.955684

Table 3.2: This table shov change in the model, as me by the loss and by the estim the parameters, after an incr amount of training epochs. T drops sharply in the first 50 tions, but even after 20000 iter there are still some changes values of the parameters.

In order to protect against over-fitting, it is common to add a check to the training loop, to say that after a minimum number of iterations has been done, we stop the training when the loss on the testing data starts increasing. In order to protect against very long training steps, it is also common to set a tolerance (absolute or relative) under which we decide that improvements to the loss are not meaningful, and which serves as a stopping criterion for the training.

3.4.4 Detecting over-fitting

As we mentioned in the previous section, one risk with training that runs for too long is to start seeing over-fitting. The usual diagnosis for over-fitting is an increase in the testing loss, which is to say, in the loss measured on the data that were not used for training. In Figure 3.2, we can see that the RMSE loss decreases at the same rate on both datasets, which indicates that the model is learning from the data, but not to a point where its ability to generalize suffers.

Underfitting is also a possible scenario, where the model is *not* learning from the data, and can be detected by seeing the loss measures remain high or even increase.

We are producting the loss over time figure after the training, as it is good practice – but as we mentionned in the previous section, it is very common to have the training code look at the dynamics of these two values in order to decide whether to stop the training early.

3.4.5 Visualizing the learning process

3.4.6 Outcome of the model

3.5 Notes on regularization

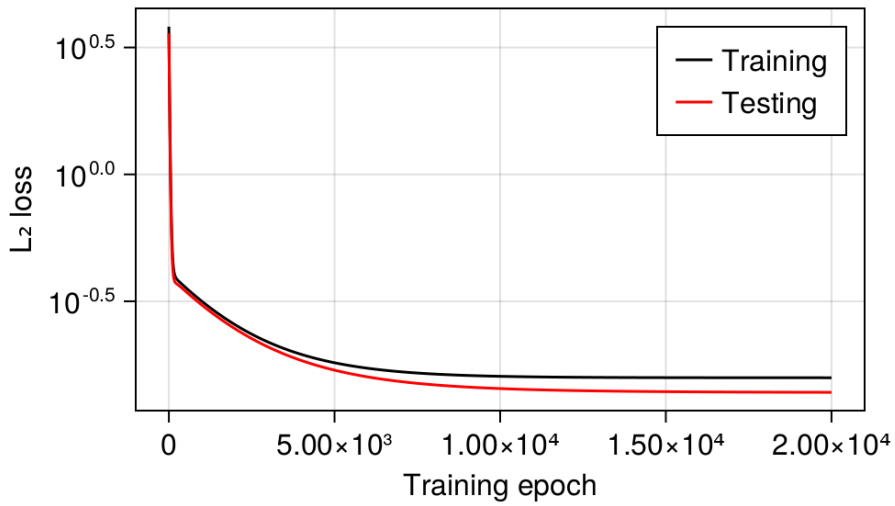

Figure 3.2: This figures shows the change in the loss for the training and testing dataset. As the two curves converge on low values at the same rate, this suggests that the model is not over-fitting, and is therefore suitable for use.

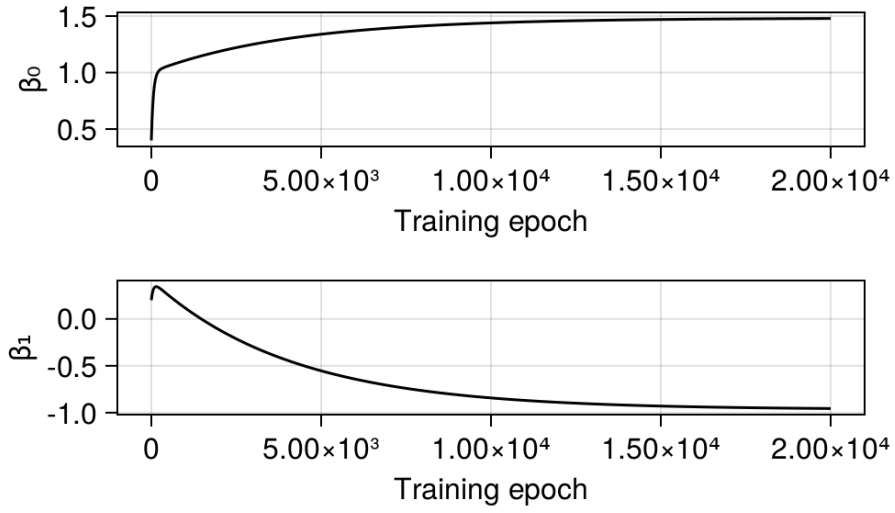

Figure 3.3: This figure shows the change in the parameters values over time. Note that the change is very large initially, because we make large steps when the gradient is strong. The rate of change gets much lower as we get nearer to the "correct" value.

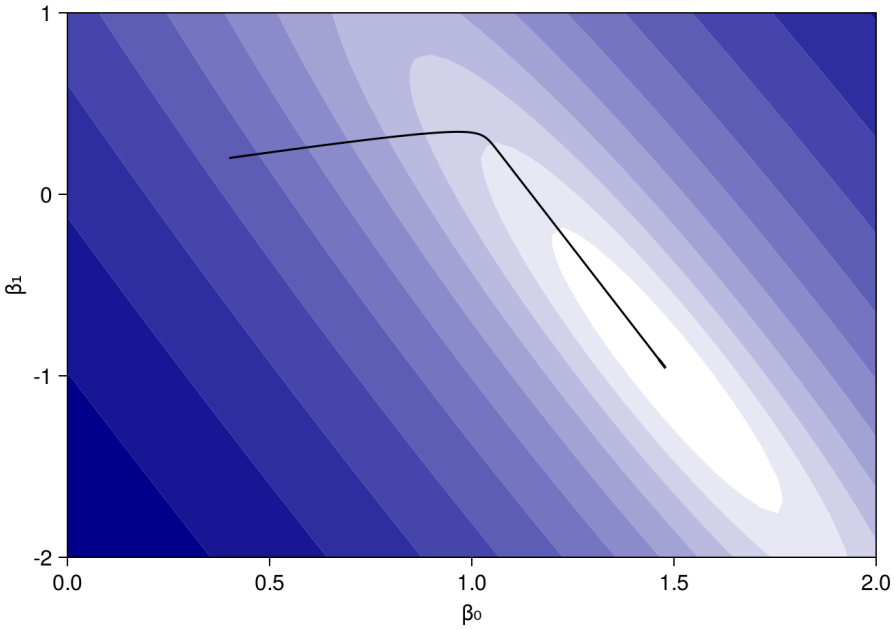

Figure 3.4: Alternative representation of the data from Figure 3.3, where the change in parameter values (solid line) is plotted on top of the surface giving the log-loss for all pairs of parameters. We can intuit this figure as showing the movement of a ball (the parameters) rolling down a bowl, at the bottom of which is the correct model parameterization.

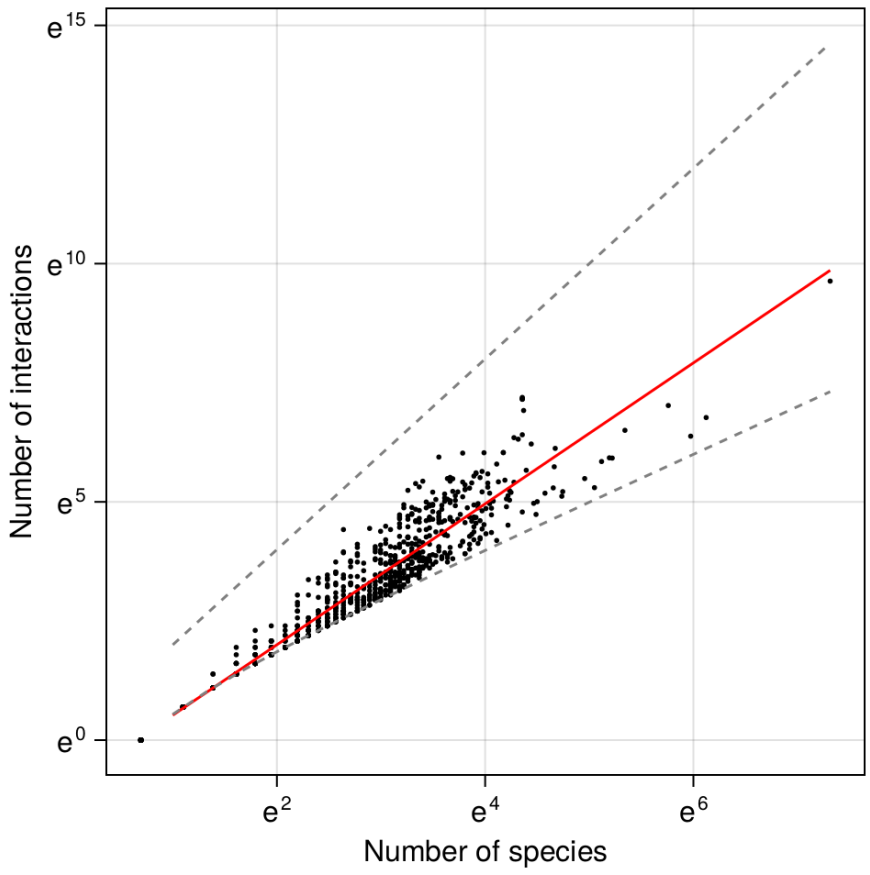

Figure 3.5: Overview of data from ...

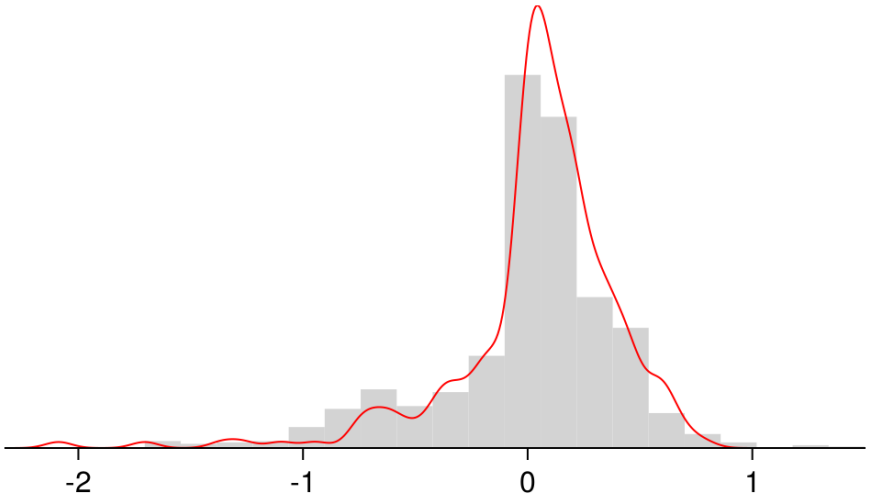

Figure 3.6: Overview of the distributions of residuals (note that the residuals are calculated on the $log_2(L)$ and $log_2(S)$) after a large enough number of epochs for gradient descent. The grey histogram is the residuals on the training data, and the red density line is the residuals on the testing data.

Chapter 4

Testing, training, validating

In Chapter 2, we were very lucky. Because we applied an unsupervised method, we didn't really have a target to compare to the output. Whatever classification we got, we had to live with it. It was incredibly freeing. Sadly, in most applications, we will have to compare our predictions to data, and data are incredibly vexatious. In this chapter, we will develop intuitions on the notions of training, testing, and validation; we will further think about data leakage, why it is somehow worse than it sounds, and how to protect against it.

In a sense, we started thinking about these concepts in Chapter 3; specifically, we came up with a way to optimize the parameters of our model (*i.e.* of *training* our model) based on a series of empirical observations, and a criteria for what a "good fit" is. We further appraised the performance of our model by measuring the loss (our measure of how good the fit is) on a dataset that was not accessible during training.

4.1 How can we split a dataset?

There is a much more important question to ask first: *why* do we split a dataset? In a sense, answering this question echoes the discussion we started in **TODO REG MOD**, because the purpose of splitting a dataset is to ensure we can train and evaluate it properly, in order to deliver the best possible model.

When a model is trained, it has learned from the data, we have tuned its hyperparameters to ensure that it learned with the best possible conditions, and we have applied a measure of performance after the entire process to communicate how well our model works. These three tasks require three different datasets, and this is the purpose of splitting our data into groups.

One of the issues when reading about splitting data is that the terminology can be muddy. For example, what constitutes a testing and validation set can

largely be a matter of perspective. In many instances, testing and validation are used interchangeably, and this book is no exception. Nevertheless, it helps to settle on a few guidelines here, before going into the details of what each dataset constitutes and how to assemble it.

The training data are examples that are given to the model during the training process. This one has no ambiguities. aside from the fact that it is defined by substraction, in a sense, as whatever is left of the original data after we set aside testing and validation sets.

The testing data are used at the end of the process, to measure the performance of a trained model with tuned hyper-parameters. If the training data are the lectures, testing data are the final exam: we can measure the performance of the model on this dataset and report it as the model performance we can expect when applying the model to new data. There is a very important, chapter-long, caveat about this last point, related to the potential of information leak between datasets, which is covered in **TODO LEAKAGE**.

The validation data are used in-between, as part of the training process. They are (possibly) a subset of the training data that we use internally to check the performance of the model, often in order to tune its hyper-parameters, or as a way to report on the over-fitting of the model during the training process.

The difference between testing and validation is largely a difference of *intent*. When we want to provide an a posteriori assessment of the model performance, the dataset we use to determine this performance is a testing dataset. When we want to optimize some aspect of the model, the data we use for this are the validation data. With this high-level perspective in mind, let's look at each of these datasets in turn. The differences between these three datasets are summarized in Table 4.1.

Dataset	Trains	Purpose	Data for performance
Training	yes	train model	
Validation		validate during training	training data only
Testing		estimates of future performance	all except testing

Table 4.1: gfkdf jdshgl dsjg d
mation in the "Data used fo
ing" column refer to the da
have been used to train the
when calculating its perform

4.1.1 Training

In data science (in applied machine learning in particular), we do *fit* models. We *train* them. This is an important difference: training is an iterative process, that we can repeat, optimize, and tweak. The outcome of training and the outcome of fitting are essentially the same (a model that is parameterized to work as well as possible on a given dataset), but it is good practice to adopt the lan-

guage of a field, and the language of data science emphasizes the different practices in model training.

Training, to provide a general definition, is the action of modifying the parameters of a model, based on knowledge of the data, and the error that results from using the current parameter values. In Chapter 3, for example, we will see how to train a linear model using the technique of gradient descent. Our focus in this chapter is not on the methods we use for training, but on the data that are required to train a model.

Training a model is a process akin to rote learning: we will present the same input, and the same expected responses, many times over, and we will find ways for the error on each response to decrease.

In order to initiate this process, we need an untrained model. Untrained, in this context, refers to a model that has not been trained *on the specific problem* we are addressing; the model may have been trained on a different problem (for example, we want to predict the distribution of a species based on a GLM trained on a phylogenetically related species). It is important to note that by "training the model", what we really mean is "change the structure of the parameters until the output looks right". For example, assuming a simple linear model like $c(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2$, training this model would lead to changes in the values of β, but not to the consideration of a new model $c(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2$. Comparing models is (often) the point of validation, which we will address later on in the book.

4.1.2 Validating

The easiest way to think about the validation dataset is by thinking about what it is *not* used for: training the model (this is the training set), and giving a final overview of the model expected performance (this is the testing set). The validation set is used for everything else (model selection, cross-validation, hyperparameters tuning), albeit in a specific way. With the training set, we communicate the predictors and the labels to the model, and update the weights of the model in response. With the validation set, we communicate the predictors and the labels to the model, but we do *not* update the weights in response. All we care about during validation is the performance of the model on a problem it has not yet encountered during training. If the training set is like attending a lecture, the validation set is like formative feedback.

Of course, one issue with the creation of a validation set is that it needs to resemble the problem the model will have to solve in practice. We will discuss this more in depth in the following sections, but it is worth thinking about an example. Assume a model that classifies a picture as having either a black bear, or no black bear. Now, we can train this model using, for example, images from 10 camera traps that are situated in a forest. And we might want to validate with a camera trap that is in a zoo. In one of the enclosures. The one with a bear. A polar one.

The issue with this dataset as a validation dataset is that is does not matches the problem we try to solve in many different ways. First, we will have an excess of images with bears compared to our problem environment. Second, the data will come from very different environments (forest v. zoo). Finally, we are attempting to validate on something that is an entirely different species of bear. This sounds like an egregious case (it is), but it is easy to commit this type of mistake when our data get more complex than black bear, polar bear, no bear.

Validation is, in particular, very difficult when the dataset we use for training has extreme events (Bellocchi *et al.* 2010). Similarly, the efficiency of validation datasets can be limited if it reflects the same biases as the training data (Martinez-Meyer 2005). Recall that this validation dataset is used to decide on the ideal conditions to train the final model before testing (and eventually, deployment); it is, therefore, extremely important to get it right. A large number of techniques to split data **REF TK** use heuristics to minimize the risk of picking the wrong validation data.

4.1.3 Testing

The testing dataset is special. The model has *never* touched it. Not during training, and not for validation. For this reason, we can give it a very unique status: it is an analogue to data that are newly collected, and ready to be passed through the trained model in order to make a prediction. The only difference between the testing set and actual new data is that, for the testing set, we know the labels. In other words, we can compare the model output to these labels, and this gives us an estimate of the model performance on future data.

But this requires a trained model, and we sort of glossed over this step. In order to come up with a trained model, it would be a strange idea not to use the validation data – they are, after all, holding information about the data we want to model! Once we have evaluated our model on the validation set, we can start the last round of training to produce the final model. We do this by training the model using everything *except* the testing data.

4.2 The problem: phenology of cherry blossom

The cherry blossom tree (*Prunus*) is ...

Long-term time series of the date of first bloom in Japan reveal that in the last decades, cherry blossom blooms earlier, which has been linked to, possibly, climate change and urbanization. The suspected causal mechanism is as follows: both global warming and urbanization lead to higher temperatures, which means a faster accumulation of degree days over the growing season, leading to an earlier bloom. Indeed, the raw data presented in Figure 4.1 show that trees bloom early when the temperatures are higher.

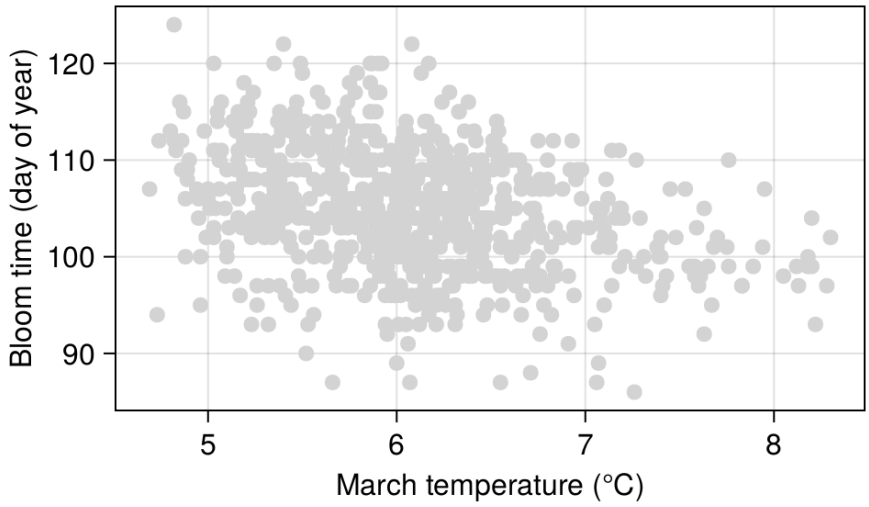

Figure 4.1: The raw data show a negative relationship between the temperature in March, and the bloom time. This suggests that when the trees have accumulated enough temperature, they can bloom early. In a context of warming, we should therefore see earlier blooms with rising temperatures.

With these data in hand (day of year with the first bloom, and smoothed reconstructed temperature in March), we can start thinking about this hypothesis. But by contrast with our simple strategy in **GRADIENT**, this time, we will split our dataset into training, validation, and testing sets, as we discussed in the previous section. Yet there are many ways to split a dataset, and therefore before starting the analysis, we will have a look at a few of them.

4.3 Strategies to split data

remove testing first if needed

exhaustive

non-exhaustive

4.3.1 Holdout

The holdout method is what we used in **TODO REF GRAD**, in which we randomly selected some observations to be part of the validation data (which was, in practice, a testing dataset in this example), and kept the rest to serve as the training data. Holdout cross-validation is possibly the simplest technique, but it suffers from a few drawbacks.

The model is only trained for one split of the data, and similarly only evaluated for one split of the data. There is, therefore, a chance to sample a particularly bad combination of the data that lead to erroneous results. Attempts to quantify the importance of the predictors are likely to give particularly unstable results,

as the noise introduced by picking a single random subset will not be smoothed out by multiple attempts.

In addition, as Hawkins *et al.* (2003) point out, holdout validation is particularly wasteful in data-limited settings, where there are fewer than hundreds of observations. The reason is that the holdout dataset will *never* contribute to training, and assuming the data are split 80/20, one out of five observations will not contribute to the model. Other cross-validation schemes presented in this section will allow observations to be used both for training and validation.

4.3.2 Leave-p-out

In leave-p-out cross-validation (LpOCV), starting from a dataset on n observation, we pick p at random to serve as validation data, and $n - p$ to serve as the training dataset. This process is then repeated *exhaustively*, which is to say we split the dataset in every possible way that gives p and $n - p$ observations, for a set value of p. The model is then trained on the $n-p$ observations, and validated on the p observations for validation, and the performance (or loss) is averaged to give the model performance before testing.

Celisse (2014) points out that p has to be large enough (relative to the sample size n) to overcome the propensity of the model to overfit on a small training dataset. One issue with LpOCV is that the number of combinations is potentially very large. It is, in fact, given by the binomial coefficent $\binom{n}{p}$, which gets unreasonably large even for small datasets. For example, running LpOCV on $n = 150$ observations, leaving out $p = 10$ for validation everytime, would require to train the model about 10^{15} times. Assuming we can train the model in 10^{-3} seconds, the entire process would require 370 centuries.

Oh well.

4.3.3 Leave-one-out

The leave-one-out cross-validation (LOOCV) is a special case of LpOCV with $p = 1$. Note that it is a lot faster to run than LpOCV, because $\binom{n}{1} = n$, and so the validation step runs in $O(n)$ (LpOCV runs in $O(n!)$). LOOCV is also an *exhaustive* cross-validation technique, as every possible way to split the dataset will be used for training and evaluation.

4.3.4 k-fold

One of the most frequent cross-validation scheme is k-fold cross-validation. Under this approach, the dataset is split into k equal parts (and so when $k = n$, this is also equivalent to LOOCV). Like with LOOCV, one desirable property of k-fold cross-validation is that each observation is used *exactly* one time to evaluate the model , and *exactly* $k - 1$ times to train it. But by contrast with the holdout validation approach, *all* observations are used to train the model.

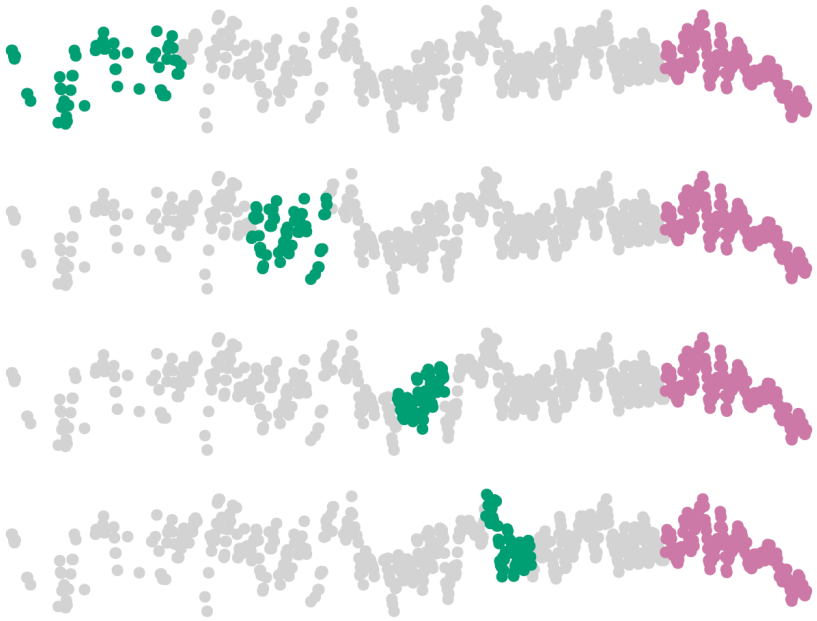

Figure 4.2: An illustration of a series of folds on a timeseries. The grey aata are used for training, the black data for validation, and the red data are kept for testing. Note that in this context, we sometimes use the future to validate on the past (look at the first fold!), but this is acceptable for reasons explained in the text.

When the data have some specific structure, it can be a good thing to manipulate the splits in order to maintain this structure. For example, Bergmeir & Benítez (2012) use temporal blocks for validation of time series, and retain the last part of the series for testing (we illustrate this in Figure 4.2). For spatial data, Hijmans (2012) suggests the use of a null model based on distance to training sites to decide on how to split the data; Valavi *et al.* (2018) have designed specific k-fold cross-validation schemes for species distribution models. These approaches all belong to the family of *stratified* k-fold cross-validation (Zeng & Martinez 2000).

The appropriate value of k is often an unknown. It is common to use $k = 10$ as a starting point (tenfold cross-validation), but other values are justifiable based on data volume, complexity of the model training, to name a few.

4.3.5 Monte-Carlo

One of the limitations of k-fold cross-validation is that the number of splits is limited by the amount of observations, especially if we want to ensure that there are enough samples in the validation data. To compensate for this, Monte-Carlo cross-validation is essentially the application (and averaging) of holdout validation an arbitrary number of times. Furthermore, the training and validation datasets can be constructed in order to account for specific constraints in the dataset, giving more flexibility than k-fold cross-validation. When the (computational) cost of training the model is high, and the dataset has specific struc-

tural constraints, Monte-Carlo cross-validation is a good way to generate data for hyperparameters tuning.

One issue with Monte-Carlo cross-validation is that we lose the guarantee that every observation will be used for training at least once (and similarly for validation). Trivially, this becomes less of an issue when we increase the number of replications, but then this suffers from the same issues as LpOCV, namely the unreasonable computational requirements.

4.4 Application: when do cherry blossom bloom?

The model we will train for this section is really simple: bloom day $= m \times$ temperature $+ b$. This is a linear model, and one with a nice, direct biological interpretation: the average (baseline) day of bloom is b, and each degree adds m days to the bloom date. At this point, we *might* start thinking about the distribution of the response, and what type of GLM we should used, but no. Not today. Today, we want to iterate quickly, and so we will start with a model that is exactly as simple as it needs to be.

This approach (start from a model that is suspiciously simple) is a good thing, for more than a few reasons. First, it gives us a baseline to compare more complicated models against. Second, it means that we do not need to focus on the complexity of the code (and the model) when building a pipeline for the analysis. Finally, and most importantly, it gives us a result very rapidly, which enables a loop of iterative model refinement on a very short timescale. Additionally, at least for this example, the simple models often work rather well.

4.4.1 Performance evaluation

We can visualize the results of our model training and assessment process. These results are presented in Figure 4.3 (as well as in Table 4.2, if you want to see the standard deviation across all splits), and follow the same color-coding convention we have used so far. All three loss measures presented here express their loss in the units of the response variable, which in this case is the day of the year where the bloom was recorded. These results show that our trained model achieves a loss of the order of a day or two in the testing data, which sounds really good!

Yet it is important to contextualize these results. What does it means for our prediction to be correct plus or minus two days? There are at least two important points to consider.

Table 4.2: TODO

Dataset	Measure	Loss (avg.)	Loss (std. dev.)
Testing	MAE	1.696	
Training	MAE	2.2397	0.0482364

Dataset	Measure	Loss (avg.)	Loss (std. dev.)
Validation	MAE	2.26331	0.421513
Testing	MBE	0.0971036	
Training	MBE	9.8278e-15	1.15597e-14
Validation	MBE	0.000419595	0.910229
Testing	MSE	4.49123	
Training	MSE	8.04855	0.32487
Validation	MSE	8.24897	2.93094
Testing	RMSE	2.11925	
Training	RMSE	2.83648	0.0570941
Validation	RMSE	2.82514	0.545232

First, what are we predicting? Our response variable is not *really* the day of the bloom, but is rather a smoothed average looking back some years, and looking ahead some years too. For this reason, we are removing a lot of the variability in the underlying time series. This is not necessarily a bad thing, especially if we are looking for a trend at a large temporal scale, but it means that we should not interpret our results at a scale lower than the duration of the window we use for averaging.

Second, what difference *does* a day make? Figure 4.1 shows that most of the days of bloom happen between day-of-year 100 and day-of-year 110. Recall that the MAE is measured by taking the average absolute error – a mistake of 24 hours is 10% of this interval! This is an example of how thinking about the units of the loss function we use for model evaluations can help us contextualize the predictions.

4.4.2 Model predictions

The predictions of our model are presented in Figure 4.4; these are the predictions of the *final* model, that is, the model that we trained on everything *except* the testing data, and for which we can get the performance by looking at Figure 4.3.

The question we now need to answer is: is our model doing a good job? We can start thinking about this question in a very qualitative way: yes, it does a goob job at drawing a line that, through time, goes right through the original data. As far as validation goes, it maybe underestimates the drop in the response variable (it predicts the bloom a little later), but maybe there are long-term effects, expressed over the lifetime of the tree (the first bloom usually takes places after 6 or 7 growth seasons), that we do not account for.

Our model tends to smooth out some of the variation; it does not predict bloom dates before day of year 100, or after day of year 108, although they do happen. This may not be a trivial under-prediction: some of these cycles leading to very early/late bloom can take place over a century, meaning that our model could

Figure 4.3: Visualisation of the model performance for .... The colors are the same as in fig-splits-illustration, *i.e.* grey for the training data, green for the validation data, and purple for the testing data.

be consistently wrong (which is to say, wrong with the same bias) for dozens of years in a row.

4.4.3 Is our model good, then?

The answer is, it depends. Models are neither good, nor bad. They are either fit, or unfit, for a specific purpose.

If the purpose is to decide when to schedule a one-day trip to see the cherry blossom bloom, our model is not really fit – looking at the predictions, it gets within a day of the date of bloom (but oh, by the way, this is an average over close to a decade!) about 15% of the time, which jumps up to close to 30% if you accept a two-days window of error.

If the purpose is to look at long-time trends in the date of bloom, then our model actually works rather well. It does under-estimate the amplitude of the cycles, but not by a large amount. In fact, we could probably stretch the predictions a little, applying a little correction factor, and have a far more interesting model.

We will often be confronted to this question when working with prediction. There is not really a criteria for "good", only a series of compromises and judgment calls about "good enough". This is important. It reinforces the imperative of keeping the practice of data science connected to the domain knowledge, as ultimately, a domain expert will have to settle on whether to use a model or not.

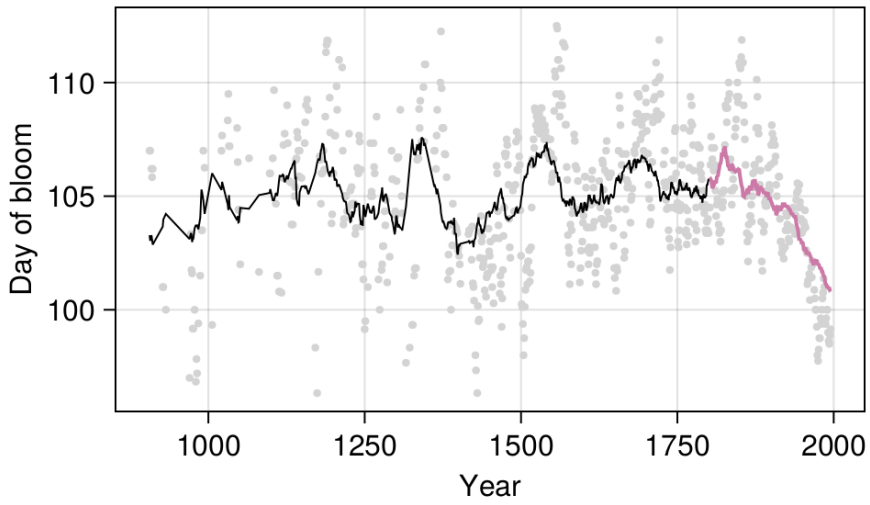

Figure 4.4: TODO

Chapter 5

Data leakage

Data leakage is a concept that is, surprisingly, grosser than it sounds. The purpose of this section is to put the fear of data leakage in you, because it can, and most assuredly *will*, lead to bad models, which is to say (as we discussed in Section 3.1), models that do not adequately represent the underlying data, in part because we have built-in some biases into them. In turn, this can eventually lead to decreased explainability of the models, which erodes trust in their predictions (Amarasinghe *et al.* 2023). As illustrated by Stock *et al.* (2023), a large number of ecological applications of machine learning are particularly susceptible to data leakage, meaning that this should be a core point of concern for us.

5.1 Consequences of data leakage

We take data leakage so seriously because it is one of the top ten mistakes in applied machine learning (Nisbet *et al.* 2018). Data leakage happens information "leaks" from the training conditions to the evaluation conditions. In other words, when the model is evaluated after mistakenly being fed information that would not be available in real-life situations. Note that this definition of leakage is different from another notion, namely the loss of data availability over time (Peterson *et al.* 2018).

It is worth stopping for a moment to consider what these "real-life situations" are, and how they differ from the training of the model. Most of this difference can be summarized by the fact that when we are *applying* a model, we can start from the model *only*. Which is to say, the data that have been used for the training and validation of the model may have been lost, without changing the applicability of the model: it works on entirely new data.

Because this is the behavior we want to simulate with a validation dataset, it is

very important to fully disconnect the validation data from the rest of the data. We can illustrate this with an example. Let's say we want to work on a time series of population size, such as provided by the *BioTIME* project (Dornelas *et al.* 2018). One naïve approach would be to split this the time series at random into three datasets. We can use one to train the models, one to test these models, and a last one for validation.

Congratulations! We have created data leakage! Because we are splitting our timeseries at random, the model will likely have been trained using data that date from *after* the start of the validation dataset. In other words: our model can peek into the future. This is highly unlikely to happen in practice, due to the laws of physics. A strategy that would prevent leakage would have been to pick a cut-off date to define the validation dataset, and then to decide how to deal with the training and testing sets.

TODO stationary processes Politis & Romano (1994)

5.2 Sources of data leakage

5.2.1 Leakage between instances

5.2.2 Leakage between features

5.2.3 Leakage during transformations

5.3 Avoiding data leakage

The most common advice given in order to prevent data leakage is the "learn/predict separation" (Kaufman *et al.* 2011). Simply put, this means that whatever happens to the data used for training cannot be *simultaneously* applied to the data used for testing (or validation).

5.3.1 An example using PCA

Assume that we want to transform our data using a Principal Component Analysis (PCA; Pearson (1901)). Ecologists often think of PCA as a technique to explore data (Legendre & Legendre 2012), but it is so much more than that! PCA is a model, because we can derive, from the data, a series of weights (in the transformation matrix), which we can then apply to other datasets in order to project them in the space of the projection of the training data.

If we have a dataset $\mathbf{X}$, which we split into two components $\mathbf{X}_0$ for training ,and $\mathbf{X}_1$ for validation, there are two ways to use a PCA to transform these data. The first is $\mathbf{T} = \mathbf{XW}$, which uses the full dataset. When we predict the position of the validation data, we could use the transformation $\mathbf{T}_1 = \mathbf{X}_1\mathbf{W}$, but this would introduce data leakage: we have trained the transformation we apply to

$\mathbf{X}_1$ using data that are already in $\mathbf{X}_1$, and therefore we have not respected the learn/predict separation.

The second way to handle this situation is to perform our PCA using $\mathbf{T}_0 = \mathbf{X}_0\mathbf{W}_0$, which is to say, the weights of our PCA are derived *only* from the training data. In this situation, whenever we project the data in the validation set using $\mathbf{T}_1 = \mathbf{X}_1\mathbf{W}_0$, we respect the learn/predict separation: the transformation of $\mathbf{X}_1$ is entirely independent from the data contained in $\mathbf{X}_1$.

5.3.2 How to generalize this approach?

Although avoiding data leakage is a tricky problem, there is a very specific mindset we can adopt that goes a long way towards not introducing it in our analyses, and it is as follows: *every data transformation step is a modeling step that is part of the learning process*.

Everything is a model that can be trained. If you want to transform a variable using the z-score, this is a model! It has two parameters that you can learn from the data, μ (the average of the variable) and σ (its standard deviation). You can apply it to a data point y with $\hat{y} = (y - \mu)\sigma^{-1}$ Because this is a model, we need a dataset to learn these parameters from, and because we want to maintain the learn/predict separation, we will use the train dataset to get the values of μ_0 and σ_0. This way, when we want to get the z-score of a new observation, for example from the testing dataset, we can get it using $\hat{y}_1 = (y_1 - \mu_0)\sigma_0^{-1}$. The data transformation is entirely coming from information that was part of the training set.

One way to get the learn/predict transformation stupendously wrong is to transform our validation, testing, or prediction data using $\hat{y}_1 = (y_1 - \mu_1)\sigma_1^{-1}$. This can be easily understood with an example. Assume that the variable y_0 is the temperature in our training dataset. We are interested in making a prediction in a world that is 2 degrees hotter, uniformly, which is to say that for whatever value of y_0, the corresponding data point we use for prediction is $y_1 = y_0 + 2$. If we take the z-score of this new value based on its own average and standard deviation, a temperature two degrees warmer in the prediction data will have the same z-score as its original value, or in other words, we have hidden the fact that there is a change in our predictors! Although this would not make a difference if we are reasonably confident that our predictors are stationary processes, there is no good reason to make this assumption.

Treating the data preparation step as a part of the learning process, which is to say that we learn every transformation on the training set, and retain this transformation as part of the prediction process, we are protecting ourselves against both data leakage and the hiding of relevant changes in our predictors.

Chapter 6

Supervised classification

In the previous chapters, we have focused on efforts on regression models, which is to say models that predict a continuous response. In this chapter, we will introduce the notion of classification, which is the prediction of a discrete variable representing a category. There are a lot of topics we need to cover before we can confidently come up with a model for classification, and so this chapter is part of a series. We will first introduce the idea of classification; in **TODO**, we will think about predictions of classes as probabilities; in **TODO**, we will generalize these ideas and think about learning curves; finally, in **TODO**, we will finally think about variables a lot more, and introduce elements of model interpretation

6.1 The problem: reindeer distribution

Throughout these chapters, we will be working on a single problem, which is to predict the distribution of the Reindeer, *Rangifer tarandus tarandus*. Species Distribution Modeling (SDM; Elith & Leathwick (2009)), or Ecological Niche Modeling (ENM), is an excellent instance of ecologist essentially doing applied machine learning already, as Beery *et al.* (2021) rightfully pointed out. In fact, the question of fitness-for-purpose, which we discussed in previous chapters, has been covered in the SDM (Guillera-Arroita *et al.* 2015). In these chapters, we will fully embrace this idea, and look at the problem of predicting where species can be as a data science problem.

These two predictors are important for plants (Clapham *et al.* 1935) or animals (Whittaker 1962) **ECOZONES**, and see *e.g.* climate change (Berteaux 2014)

data from (Fick & Hijmans 2017), an alternative is (Karger *et al.* 2017) which is higher resolution, so for iteration we work with a smaller data product

6.2 What is classification?

Classification is the prediction of a qualitative response. In Chapter 2, for example, we predicted the class of a pixel, which is a qualitative variable with levels $\{1, 2, \dots, k\}$. This represented an instance of *unsupervised* learning, as we had no *a priori* notion of the correct class of the pixel. When building SDMs, by contrast, we often know where species are, and we can simulate "background points", that represent assumptions about where the species are not (Barbet-Massin *et al.* 2012; Hanberry *et al.* 2012).

In short, our response variable has levels $\{0, 1\}$: the species is there, or it is not – we will challenge this assumption later in the series of chapters, but for now, this will do.

6.2.1 Separability and the curse of dimensionality

A very important feature of the relationship between the features and the classes is that, broadly speaking, classification is much easier when the classes are separable. Separability (often linear separability) is achieved when, if looking at some projection of the data on two dimensions, you can draw a line that separates the classes (a point in a single dimension, a plane in three dimension, and so on and so forth).

In Figure 6.1, we can see the temperature (in degrees) for locations with recorded presences of reindeers (top), and for locations with assumed absences. These two classes are not quite linearly separable alongside this single dimension (but maybe there is a different projection of the data that would change this), but there are still some values at which our guess for a class changes. For example, at a location with a temperature colder than 0°C, presences are far more likely. For a location with a temperature warmer than 5°C, absences become more likely. The locations with a temperature between 0°C and 5°C can go either way.

It would be tempting to say that adding dimensions should improve our chances to find a feature alongside which the classes become linearly separable. If only!

curse of dimensionality

6.2.2 The confusion table

The most important element we need in order to evaluate the performance of a classification model is the confusion table.

$$\begin{pmatrix} \text{TP} & \text{FP} \\ \text{FN} & \text{TN} \end{pmatrix}$$

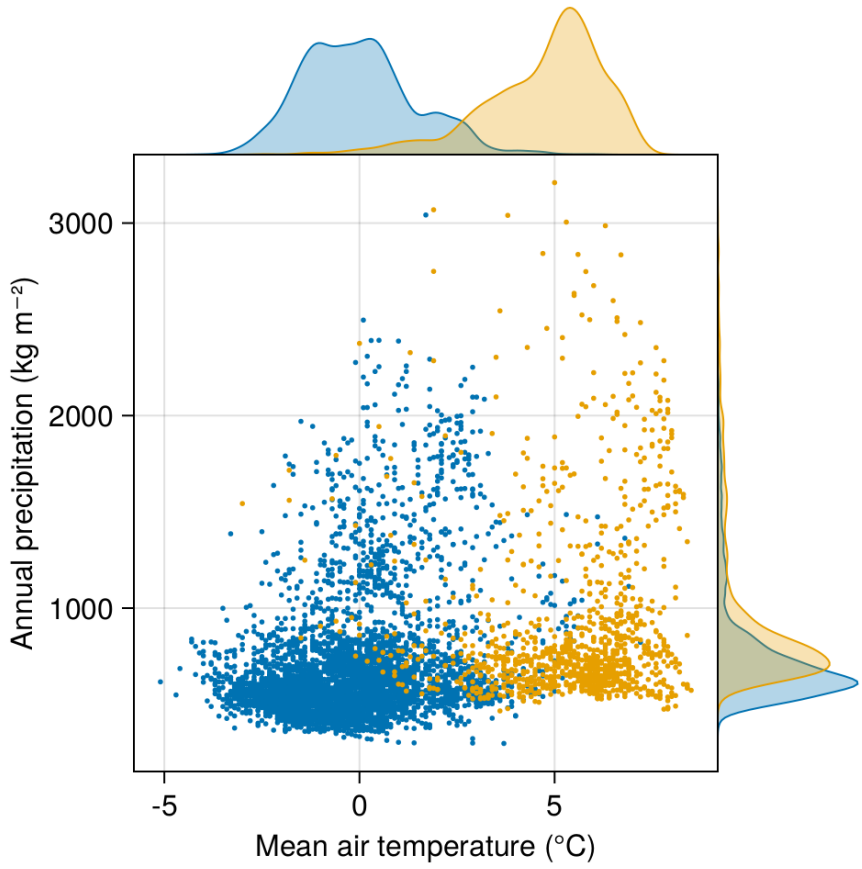

Figure 6.1: This figures show the separability of the presences (blue) and pseudo-absences (orange) on the temperature, but not on the precipitation, dimensions.

6.3 The Naive Bayes Classifier

The Naive Bayes Classifier (NBC) is my all-time favorite classifier. It is build on a very simple intuition, works with almost no data, and more importantly, often provides an annoyingly good baseline for other, more complex classifiers to meet. That NBC works at all is counter-intuitive (Hand & Yu 2001). It assumes that all variables are independent, it works when reducing the data to a simpler distribution, and although the numerical estimate of the class probability is remarkably unstable, it generally gives good predictions. NBC is the data science equivalent of saying "eh, I reckon it's probably *this* class" and somehow getting it right 95% of the case.

6.3.1 Assumptions of the NBC

In Figure 6.1, what is the most likely class if the temperature is 2°C?

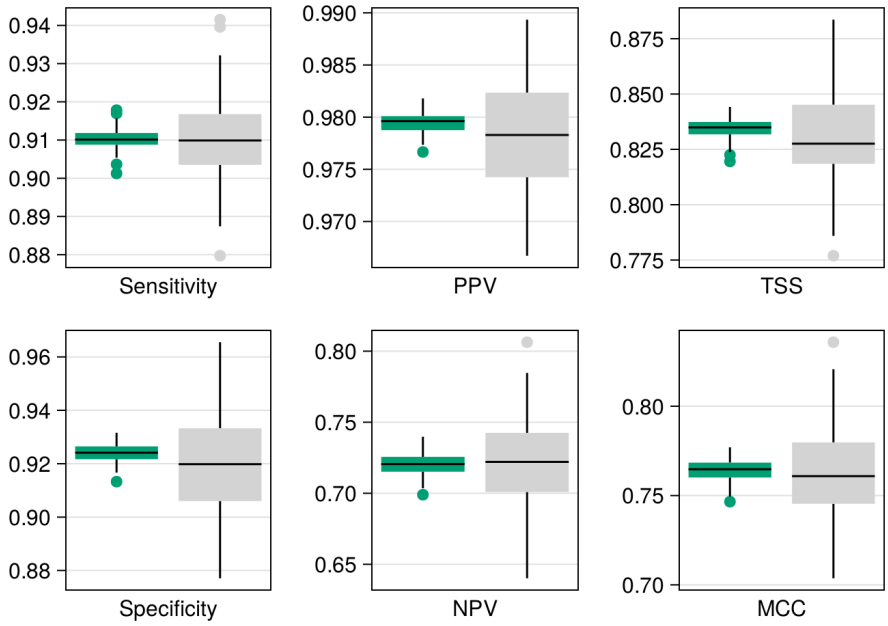

Figure 6.2: WAH NEVER CHANGES

6.3.2 How to train the NBC

6.3.3 Decision rule for prediction

6.4 Application: a baseline model of reindeer presence

6.4.1 Training and validation strategy

6.4.2 Performance evaluation of the model

6.4.3 The decision boundary

Now that the model is trained, we can take a break in our discussion of its performance, and think about *why* it makes a specific classification in the first place. Because we are using a model with only two input features, we can generate a grid of variables, and the ask, for every point on this grid, the classification made by our trained model. This will reveal the regions in the space of parameters where the model will conclude that the species is present.

The output of this simulation is given in Figure 6.3. Of course, in a model with more features, we would need to adapt our visualisations, but because we only use two features here, this image actually gives us a complete understanding of the model decision process. Think of it this way: even if we lose the code of the model, we could use this figure to classify any input made of a temperature

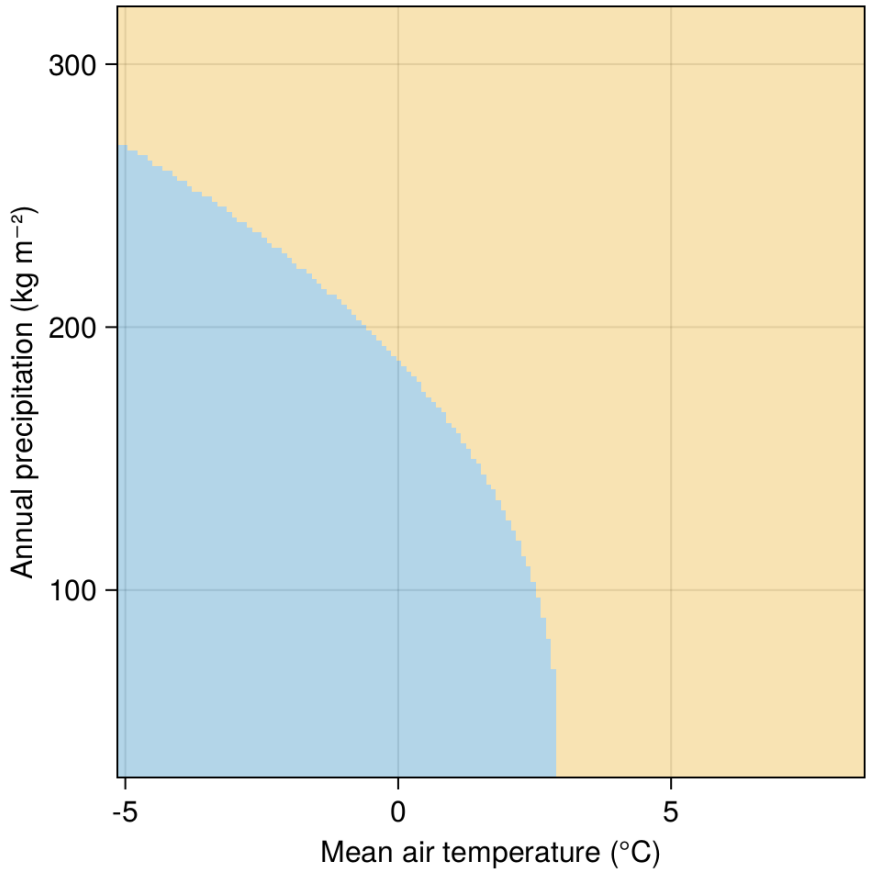

Figure 6.3: Overview of the decision boundary between the positive (blue) and negative (classes) using the NBC with two variables. Note that, as expected with a Gaussian distribution, the limit between the two classes looks circular. The assumption of statistical independance between the features means that we would not see, for example, an ellipse.

and a precipitation, and read what the model decision would have been.

The line that separates the two classes is usually refered to as the "decision boundary" of the classifier: crossing this line by moving in the space of features will lead the model to predict another class at the output. In this instance, as a consequence of the choice of models and of the distribution of presence and absences in the environmental space, the decision boundary is not linear.

It is interesting to compare Figure 6.3 with, for example, the distribution of the raw data presented in Figure 6.1. Although we initially observed that temperature was giving us the best chance to separate the two classes, the shape of the decision boundary suggests that our classifier is considering that reindeers enjoy cold and dry climates.

6.4.4 What is an acceptable model?

Chapter 7

Variable selection

we will use MCC (Chicco & Jurman 2020) rather than F_1 or accuracy to decide on the best model

is the Pearson product-moment correlation on a contingency table (Powers 2020)

Chapter 8

Learning curves and moving thresholds

In **TK GRADIENT**, we represented the testing and training loss of a model as a function of the number of gradient descent steps we had made. This sort of representation is very useful to figure out how well our model is learning, and is called, appropriately enough, a learning curve. In this chapter, we will produce learning curves to find the optimal values of hyper-parameters of our model. We will illustrate this using an approach called moving-threshold classification, and additionally explore how we can conduct grid searches to tune several hyper-parameters at once.

8.1 Classification based on probabilities

When first introducing classification, in **TODO CHAPTER**, we used a model that returned a deterministic answer, which is to say, the name of a class (in our case, this class was either "present" or "absent"). But a lot of classifiers return quantitative values, that correspond to (proxies for) the probability of the different classes. Nevertheless, because we are interesting in solving a classification problem, we need to end up with a confusion table, and so we need to turn a number into a class. In the context of binary classification (we model a yes/no variable), this can be done using a threshold for the probability.

The idea behind the use of thresholds is simple: if the classifier output $\hat{y}$ is larger than (or equal to) the threshold value τ, we consider that this prediction corresponds to the positive class (the event we want to detect, for example the presence of a species). In the other case, this prediction corresponds to the negative class. Note that we do not, strictly, speaking, require that the value $\hat{y}$ returned by the classifier be a probability. We can simply decide to pick τ somewhere in the support of the distribution of $\hat{y}$.

The threshold to decide on a positive event is an hyper-parameter of the model. In the NBC we built in **TODO**, our decision rule was that $p(+) > p(-)$, which when all is said and done (but we will convince ourselves of this in **TODO**), means that we used $\tau = 0.5$. But there is no reason to assume that the threshold needs to be one half. Maybe the model is overly sensitive to negatives. Maybe there is a slight issue with our training data that bias the model predictions. And for this reason, we have to look for the optimal value of τ.

There are two important values for the threshold, at which we know the behavior of our model. The first is $\tau = \min(\hat{y})$, for which the model *always* returns a negative answer; the second is, unsurprisingly, $\tau = \max(\hat{y})$, where the model *always* returns a positive answer. Thinking of this behavior in terms of the measures on the confusion matrix, as we have introduced them in **TODO**, the smallest possible threshold gives only negatives, and the largest possible one gives only positives: they respectively maximize the false negatives and false positives rates.

8.1.1 The ROC curve

This is a behavior we can exploit, as increasing the threshold away from the minimum will lower the false negatives rate and increase the true positive rate, while decreasing the threshold away from the maximum will lower the false positives rate and increase the true negative rate. If we cross our fingers and knock on wood, there will be a point where the false events rates have decreased as much as possible, and the true events rates have increased as much as possible, and this corresponds to the optimal value of τ for our problem.

We have just described the Receiver Operating Characteristic (ROC; Fawcett (2006)) curve! The ROC curve visualizes the false positive rate on the x axis, and the true positive rate on the y axis. The area under the curve (the ROC-AUC) is a measure of the overall performance of the classifier (Hanley & McNeil 1982); a model with ROC-AUC of 0.5 performs at random, and values moving away from 0.5 indicate better (close to 1) or worse (close to 0) performance.The ROC curve is a description of the model performance across all of the possible threshold values we investigated!

8.1.2 The PR curve

One very common issue with ROC curves, is that they are overly optimistic about the performance of the model, especially when the problem we work on suffers from class imbalance, which happens when observations of the positive class are much rarer than observations of the negative class. In ecology, this is a common feature of data on species interactions (Poisot *et al.* 2023). For this reason, it is always advised to ...

8.1.3 Cross-entropy loss and other classification loss functions

8.2 How to optimize the threshold?

In order to understand the optimization of the threshold, we first need to understand how a model with thresholding works. When we run such a model on multiple input features, it will return a list of probabilities, for example $[0.2, 0.8, 0.1, 0.5, 1.0]$. We then compare all of these values to an initial threshold, for example $\tau = 0.05$, giving us a vector of Boolean values, in this case $[+, +, +, +, +]$. We can then compare this classified output to a series of validation labels, *e.g.* $[-, +, -, -, +]$, and report the performance of our model. In this case, the very low thresholds means that we accept any probability as a positive case, and so our model is very strongly biased. We then increase the threshold, and start again.

As we have discussed in **TODO previous**, moving the threshold is essentially a way to move in the space of true/false rates. As the measures of classification performance capture information that is relevant in this space, there should be a value of the threshold that maximizes one of these measures. Alas, no one agrees on which measure this should be (Perkins & Schisterman 2006; Unal 2017). The usual recommendation is to use the True Skill Statistic, also known as Youden's J (Youden 1950). The biomedical literature, which is quite naturally interested in getting the interpretation of tests right, has established that maximizing this value brings us very close to the optimal threshold for a binary classifier (Perkins & Schisterman 2005). In a simulation study, using the True Skill Statistic gave good predictive performance for models of species interactions (Poisot 2023).

Some authors have used the MCC as a measure of optimality (Zhou & Jakobsson 2013), as it is maximized *only* when a classifier gets a good score for the basic rates of the confusion matrix. Based on this information, Chicco & Jurman (2023) recommend that MCC should be used to pick the optimal threshold *regardless of the question*, and I agree with their assessment. A high MCC is always associated to a high ROC-AUC, TSS, etc., but the opposite is not necessarily true. This is because the MCC can only reach high values when the model is good at *everything*, and therefore it is not possible to trick it. In fact, previous comparisons show that MCC even outperform measures of classification loss (Jurman *et al.* 2012).

For once, and after over 15 years of methodological discussion, it appears that we have a conclusive answer! In order to pick the optimal threshold, we find the value that maximizes the MCC. Note that in previous chapters, we already used the MCC as a our criteria for the best model, and now you know why.

8.3 The problem: building a probabilistic NBC model

8.4 Application: improved reindeer distribution model

915	20
21	198

Table 8.1: confusion table aft
ing the value of τ etc etc

References

Amarasinghe, K., Rodolfa, K.T., Lamba, H. & Ghani, R. (2023). Explainable machine learning for public policy: Use cases, gaps, and research directions. *Data & Policy*, 5.

Barbet-Massin, M., Jiguet, F., Albert, C.H. & Thuiller, W. (2012). Selecting pseudo-absences for species distribution models: how, where and how many? *Methods in Ecology and Evolution*, 3, 327–338.

Beery, S., Cole, E., Parker, J., Perona, P. & Winner, K. (2021). Species distribution modeling for machine learning practitioners: A review. *ACM SIGCAS Conference on Computing and Sustainable Societies* (*COMPASS*).

Bellocchi, G., Rivington, M., Donatelli, M. & Matthews, K. (2010). Validation of biophysical models: issues and methodologies. A review. *Agronomy for Sustainable Development*, 30, 109–130.

Bergmeir, C. & Benítez, J.M. (2012). On the use of cross-validation for time series predictor evaluation. *Information Sciences*, 191, 192–213.

Berteaux, D. (2014). *Changements climatiques et biodiversité du québec*. Presses de l'Université du Québec.

Bezanson, J., Edelman, A., Karpinski, S. & Shah, V.B. (2017). Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59, 65–98.

Blaom, A., Kiraly, F., Lienart, T., Simillides, Y., Arenas, D. & Vollmer, S. (2020). MLJ: A julia package for composable machine learning. *Journal of Open Source Software*, 5, 2704.

Brose, U., Ostling, A., Harrison, K. & Martinez, N.D. (2004). Unified spatial scaling of species and their trophic interactions. *Nature*, 428, 167–171.

Celisse, A. (2014). Optimal cross-validation in density estimation with the l^{2}-loss. *The Annals of Statistics*, 42.

Chicco, D. & Jurman, G. (2020). The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC Genomics*, 21.

Chicco, D. & Jurman, G. (2023). The Matthews correlation coefficient (MCC) should replace the ROC AUC as the standard metric for assessing binary classification. *BioData Mining*, 16.

Clapham, A.R., Raunkiaer, C., Gilbert-Carter, H., Tansley, A.G. & Fausboll. (1935). The life forms of plants and statistical plant geography. *The Journal of Ecology*, 23, 247.

Cohen, J.E. & Briand, F. (1984). Trophic links of community food webs. *Proc Natl Acad Sci U S A*, 81, 4105–4109.

Cooney, C.R., He, Y., Varley, Z.K., Nouri, L.O., Moody, C.J.A., Jardine, M.D., *et al.* (2022). Latitudinal gradients in avian colourfulness. *Nature Ecology & Evolution*, 6, 622–629.

Cooper, N., Bond, A.L., Davis, J.L., Portela Miguez, R., Tomsett, L. & Helgen, K.M. (2019). Sex biases in bird and mammal natural history collections. *Proceedings of the Royal Society B: Biological Sciences*, 286, 20192025.

Dornelas, M., Antão, L.H., Moyes, F., Bates, A.E., Magurran, A.E., Adam, D., *et al.* (2018). BioTIME: A database of biodiversity time series for the Anthropocene. *Global Ecology and Biogeography*, 27, 760–786.

Elith, J. & Leathwick, J.R. (2009). Species Distribution Models: Ecological Explanation and Prediction Across Space and Time. *Annual Review of Ecology, Evolution, and Systematics*, 40, 677–697.

Fawcett, T. (2006). An introduction to ROC analysis. *Pattern Recognition Letters*, 27, 861–874.

Fick, S.E. & Hijmans, R.J. (2017). WorldClim 2: new 1-km spatial resolution climate surfaces for global land areas. *International Journal of Climatology*, 37, 4302–4315.

Gorman, K.B., Williams, T.D. & Fraser, W.R. (2014). Ecological Sexual Dimorphism and Environmental Variability within a Community of Antarctic Penguins (Genus Pygoscelis). *PLoS ONE*, 9, e90081.

Guillera-Arroita, G., Lahoz-Monfort, J.J., Elith, J., Gordon, A., Kujala, H., Lentini, P.E., *et al.* (2015). Is my species distribution model fit for purpose? Matching data and models to applications. *Global Ecology and Biogeography*, 24, 276–292.

Hanberry, B.B., He, H.S. & Palik, B.J. (2012). Pseudoabsence Generation Strategies for Species Distribution Models. *PLoS ONE*, 7, e44486.

Hand, D.J. & Yu, K. (2001). Idiot's bayes: Not so stupid after all? *International Statistical Review / Revue Internationale de Statistique*, 69, 385.

Hanley, J.A. & McNeil, B.J. (1982). The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology*, 143, 29–36.

Hawkins, D.M., Basak, S.C. & Mills, D. (2003). Assessing Model Fit by Cross-Validation. *Journal of Chemical Information and Computer Sciences*, 43, 579–586.

Hijmans, R.J. (2012). Cross-validation of species distribution models: removing spatial sorting bias and calibration with a null model. *Ecology*, 93, 679–688.

Innes, M. (2018). Don't unroll adjoint: Differentiating SSA-form programs.

Jurman, G., Riccadonna, S. & Furlanello, C. (2012). A Comparison of MCC and CEN Error Measures in Multi-Class Prediction. *PLoS ONE*, 7, e41882.

Karger, D.N., Conrad, O., Böhner, J., Kawohl, T., Kreft, H., Soria-Auza, R.W., *et al.* (2017). Climatologies at high resolution for the earth's land surface areas. *Scientific Data*, 4.

Kaufman, S., Rosset, S. & Perlich, C. (2011). Leakage in data mining. *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*.

Kennedy, S. & Burbach, M. (2020). Great Plains Ranchers Managing for Vege-

tation Heterogeneity: A Multiple Case Study. *Great Plains Research*, 30, 137–148.

Legendre, P. & Legendre, L. (2012). *Numerical ecology*. Developments in environmental modelling. Third English edition. Elsevier, Oxford, UK.

Luccioni, A.S. & Rolnick, D. (2023). Bugs in the data: How ImageNet misrepresents biodiversity. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37, 14382–14390.

MacDonald, A.A.M., Banville, F. & Poisot, T. (2020). Revisiting the links-species scaling relationship in food webs. *Patterns*, 1.

Martinez, N.D. (1992). Constant connectance in community food webs. *The American Naturalist*, 139, 1208–1218.

Martinez-Meyer, E. (2005). Climate change and biodiversity: Some considerations in forecasting shifts in species' potential distributions. *Biodiversity Informatics*, 2.

Nisbet, R., Miner, G., Yale, K., Elder, J.F. & Peterson, A.F. (2018). *Handbook of statistical analysis and data mining applications*. Second edition. Academic Press, London.

Pearson, K. (1901). LIII. *On lines and planes of closest fit to systems of points in space*. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2, 559–572.

Perkins, N.J. & Schisterman, E.F. (2005). The Youden Index and the Optimal Cut-Point Corrected for Measurement Error. *Biometrical Journal*, 47, 428–441.

Perkins, N.J. & Schisterman, E.F. (2006). The Inconsistency of "Optimal" Cut-points Obtained using Two Criteria based on the Receiver Operating Characteristic Curve. *American Journal of Epidemiology*, 163, 670–675.

Peterson, A.T., Asase, A., Canhos, D., Souza, S. de & Wieczorek, J. (2018). Data leakage and loss in biodiversity informatics. *Biodiversity Data Journal*, 6.

Poisot, T. (2023). Guidelines for the prediction of species interactions through binary classification. *Methods in Ecology and Evolution*, 14, 1333–1345.

Poisot, T., Ouellet, M.-A., Mollentze, N., Farrell, M.J., Becker, D.J., Brierley, L., *et al.* (2023). Network embedding unveils the hidden interactions in the mammalian virome. *Patterns*, 4, 100738.

Politis, D.N. & Romano, J.P. (1994). The Stationary Bootstrap. *Journal of the American Statistical Association*, 89, 1303–1313.

Powers, D.M.W. (2020). Evaluation: From precision, recall and f-measure to ROC, informedness, markedness and correlation. *arXiv*.

Stock, A., Gregr, E.J. & Chan, K.M.A. (2023). Data leakage jeopardizes ecological applications of machine learning. *Nature Ecology & Evolution*.

Tuia, D., Kellenberger, B., Beery, S., Costelloe, B.R., Zuffi, S., Risse, B., *et al.* (2022). Perspectives in machine learning for wildlife conservation. *Nature Communications*, 13, 792.

Unal, I. (2017). Defining an Optimal Cut-Point Value in ROC Analysis: An Alternative Approach. *Computational and Mathematical Methods in Medicine*, 2017, 1–14.

Valavi, R., Elith, J., Lahoz-Monfort, J.J. & Guillera-Arroita, G. (2018). block CV : An r package for generating spatially or environmentally separated folds

for *k* -fold cross-validation of species distribution models. *Methods in Ecology and Evolution*, 10, 225–232.

Vermote, E., Justice, C., Claverie, M. & Franch, B. (2016). Preliminary analysis of the performance of the Landsat 8/OLI land surface reflectance product. *Remote Sensing of Environment*, 185, 46–56.

Whittaker, R.H. (1962). Classification of natural communities. *The Botanical Review*, 28, 1–239.

Youden, W.J. (1950). Index for rating diagnostic tests. *Cancer*, 3, 32–35.

Zeng, X. & Martinez, T.R. (2000). Distribution-balanced stratified cross-validation for accuracy estimation. *Journal of Experimental & Theoretical Artificial Intelligence*, 12, 1–12.

Zhou, H. & Jakobsson, E. (2013). Predicting Protein-Protein Interaction by the Mirrortree Method: Possibilities and Limitations. *PLoS ONE*, 8, e81100.