Interpretable machine learning for species distribution modeling

CSEE 2025 Workshop

Michael D. Catchen Timothée Poisot
Gabriel Dansereau
Ariane Bussières-Fournel

To do.

TODO LIST

- Decide on species to align with abstract
- Boosting merged and utilized
- RF w/ thresholding gives map w/ 0s, 1s, and a small number of 0.5s
- `mask(presences, poly)` requires rewrapping it in an Occurrence object. See no downside in always doing this by default?

Setup Packages

```
using Pkg
Pkg.activate(@__DIR__)
using Random
Random.seed!(1234567)
```

```
  Activating project at `~/Code/Workspaces/InterpretableSDMs_CSEE2025`
```

```
TaskLocalRNG()
```

```
using SpeciesDistributionToolkit
using CairoMakie
using Dates
using PrettyTables
using Statistics
using DataFrames
const SDT = SpeciesDistributionToolkit
```

```
SpeciesDistributionToolkit
```

Getting Started

In this tutorial, we are going to build a species distribution model (SDM) for the species *Turdus torquatus*, a European thrush that has breeding grounds in the French, Swiss, and Italian alps. We'll particularly only focus on modeling its range in Switzerland, to present `SpeciesDistributionToolkit`'s abilities for working with polygons for countries and their subdivisions.

SpeciesDistributionToolkit.jl (SDT) is a package we (the authors) have developed over several years for working with species distribution data, fitting SDMs with various machine learning methods, and tools for interpreting these models. You can read the (recently acceptedin PCI Ecology) preprint here.

Specifically, **SDT** is a *monorepo*, composed of many subpackages — the ones we'll use here are:

- **SimpleSDMLayers**: for manipulating raster data
- **SimpleSDMDatasets**: for querying and downloading raster data from a variety of databases
- **SimpleSDMPolygons**: for querying and manipulating polygon data from a variety of databases

- **GBIF**: for downloading data from the Global Biodiversity Information Facility
- **Phylopic.jl**: for downloading taxon silhouettes to add to data visualization
- **SDeMo.jl**: a high-level interface for training, validating, and interpreting species distribution models

Downloading Polygon Data

Let's start by downloading a polygon for the border of Switzerland.

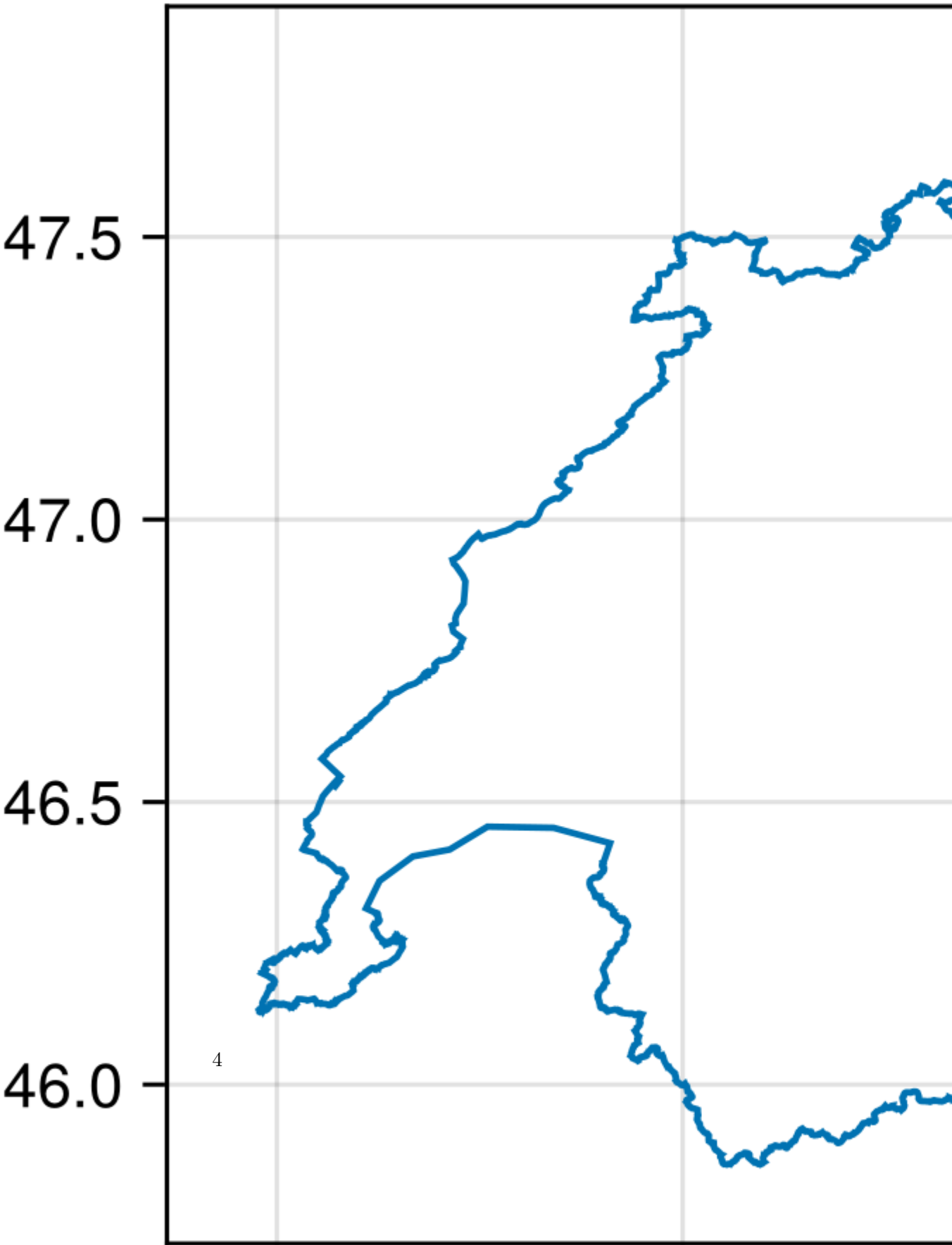
```
osm_provider = PolygonData(OpenStreetMap, Places)
switzerland = getpolygon(osm_provider, place="Switzerland")
```

```
FeatureCollection with 1 features, each with 0 properties
```

We can confirm we downloaded the right data by visualizing it. All visualizations in this tutorial will use the `Makie` library[1].

```
lines(switzerland)
```

[1] Makie uses various "backends" to produce figures, depending on the desired output format. We ran `using CairoMakie` to use the `Cairo` backend, which is used for producing publication quality PNGs and vector graphics.


47.5
47.0
46.5
46.0

Looks good! Now we'll load the environmental data we'll use.

Downloading Environmental Data

We'll download environment data from CHELSA, which provides 19 bioclimatic layers at 1km^2 resolution. The interface for downloading raster data is similar to polygon data — first we define a `RasterData` provider[^rasterdata] which takes in the database (`CHELSA2`) and particular dataset (`BioClim`) to download.

[^2]: A full list of databases, and the datasets they provide, is available here, and in the "Datasets" tab in the top navigation bar.

[^rasterdata] [2] rasterdata

```
chelsa_provider = RasterData(CHELSA2, BioClim)
```

```
RasterData{CHELSA2, BioClim}(CHELSA2, BioClim)
```

We can use the `layerdescriptions` method to list the names of all of the layers provided by `chelsa_provider`, along with their descriptions.

```
layerdescriptions(chelsa_provider)
```

```
Dict{String, String} with 19 entries:
  "BIO8"  => "Mean Temperature of Wettest Quarter"
  "BIO14" => "Precipitation of Driest Month"
  "BIO16" => "Precipitation of Wettest Quarter"
  "BIO18" => "Precipitation of Warmest Quarter"
  "BIO19" => "Precipitation of Coldest Quarter"
  "BIO10" => "Mean Temperature of Warmest Quarter"
  "BIO12" => "Annual Precipitation"
  "BIO13" => "Precipitation of Wettest Month"
  "BIO2"  => "Mean Diurnal Range (Mean of monthly (max temp - min temp))"
  "BIO11" => "Mean Temperature of Coldest Quarter"
  "BIO6"  => "Min Temperature of Coldest Month"
  "BIO4"  => "Temperature Seasonality (standard deviation ×100)"
  "BIO17" => "Precipitation of Driest Quarter"
```

```
"BIO7"  => "Temperature Annual Range (BIO5-BIO6)"
"BIO1"  => "Annual Mean Temperature"
"BIO5"  => "Max Temperature of Warmest Month"
"BIO9"  => "Mean Temperature of Driest Quarter"
"BIO3"  => "Isothermality (BIO2/BIO7) (×100)"
"BIO15" => "Precipitation Seasonality (Coefficient of Variation)"
```

To download a layer, we use the `SDMLayer` constructor, and pass the specific name of the we want `layer` keyword argument. We also pass the bounding box of the region we want with the `left`, `right`, `bottom` and `top` keywords.

For example, to download BIO1 (mean annual temperature) at longitudes from 40° to 43°, and latitudes from 30° to 35°, we run

```
SDMLayer(chelsa_provider, layer="BIO1", left=40, right=43, bottom=30, top=35)
```

```
A 601 × 361 layer with 216961 UInt16 cells
 Projection: +proj=longlat +datum=WGS84 +no_defs
```

We want to download each layer for the bounding box of Switzerland. We can obtain this by using the `boundingbox` method[3]

[3] Note that we use `SDT.boundingbox` because `boundingbox` shares a name with another method in `CairoMakie`, so we have to specificy which `boundingbox` method we mean.

```
SDT.boundingbox(switzerland)
```

```
(left = 5.955911159515381, right = 10.492294311523438, bottom = 45.81795883178711, top = 47.808
```

Note that this returns a named-tuple with each coordinate named in the same way we need them for downloading a layer. This allows us to directly input the result of `SDT.boundingbox` into `SDMLayer` using splatting[4], e.g.

[4] *Splatting* refers to adding `...` after a collection of items (like a vector or tuple), which results in them each being processed as sequential arguments. For example, if `x=[1,2,3]` and you call `f(x...)`, this is equivalent to `f(1,2,3)`.

```
SDMLayer(chelsa_provider; layer="BIO1", SDT.boundingbox(switzerland)...)
```

```
A 240 × 546 layer with 131040 UInt16 cells
 Projection: +proj=longlat +datum=WGS84 +no_defs
```

We can then load all 19 bioclimatic variables by using `layers`, which returns a list of the names of each layer provided by `chelsa_provider`, e.g.

```
layers(chelsa_provider)
```

```
19-element Vector{String}:
 "BIO1"
 "BIO2"
 "BIO3"
 "BIO4"
 "BIO5"
 "BIO6"
 "BIO7"
 "BIO8"
 "BIO9"
 "BIO10"
 "BIO11"
 "BIO12"
 "BIO13"
 "BIO14"
 "BIO15"
 "BIO16"
 "BIO17"
 "BIO18"
 "BIO19"
```

We can then load them all in a single line using an in-line for loop.

> **Note on downloading and storing layers**
>
> Note that the first time you run the following lines, the entirety of each layer will be downloaded and cached. This means the first time you run this line, it will take several minutes, but every subsequent time will be nearly instant, because the layers are saved in SimpleSDMDataset's cache (by default, this is located in `~/.julia/SimpleSDMDatasets/`).

```
env_covariates = SDMLayer{Float32}[
    SDMLayer(
        chelsa_provider;
        layer = layername,
        SDT.boundingbox(switzerland)...
    )
    for layername in layers(chelsa_provider)
]
```

```
19-element Vector{SDMLayer{Float32}}:
    A 240 × 546 layer (131040 Float32 cells)
    A 240 × 546 layer (131040 Float32 cells)
    A 240 × 546 layer (131040 Float32 cells)
    A 240 × 546 layer (131040 Float32 cells)
    A 240 × 546 layer (131040 Float32 cells)
    A 240 × 546 layer (131040 Float32 cells)
    A 240 × 546 layer (131040 Float32 cells)
    A 240 × 546 layer (131040 Float32 cells)
    A 240 × 546 layer (131040 Float32 cells)
    A 240 × 546 layer (131040 Float32 cells)
    A 240 × 546 layer (131040 Float32 cells)
    A 240 × 546 layer (131040 Float32 cells)
    A 240 × 546 layer (131040 Float32 cells)
    A 240 × 546 layer (131040 Float32 cells)
    A 240 × 546 layer (131040 Float32 cells)
    A 240 × 546 layer (131040 Float32 cells)
    A 240 × 546 layer (131040 Float32 cells)
    A 240 × 546 layer (131040 Float32 cells)
    A 240 × 546 layer (131040 Float32 cells)
```

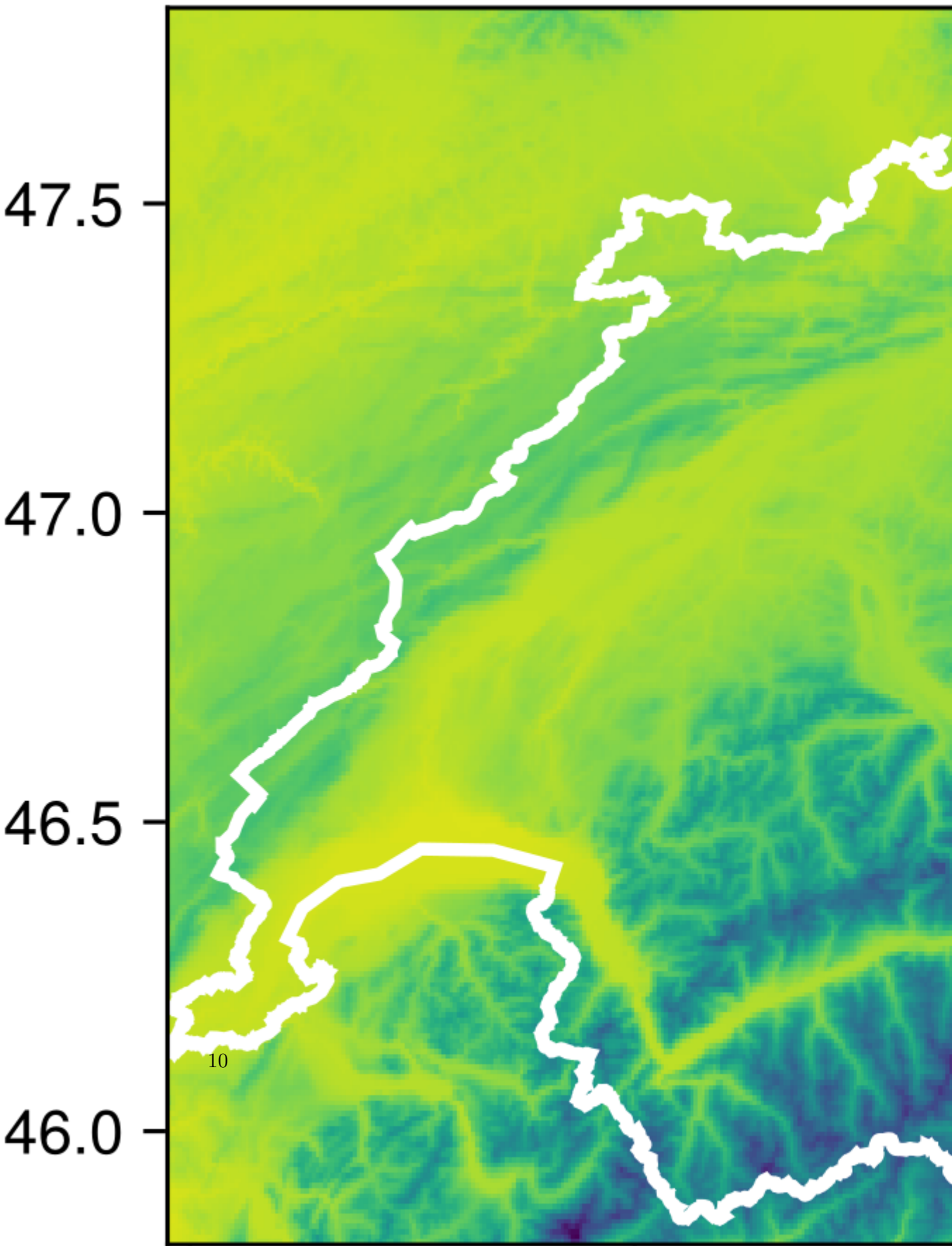
> 💡 Other ways to iterate over the layers we want to download
>
> Note that there are many different ways to do iteration in Julia, not just the in-line for loop used above.

```
env_covariates = SDMLayer{Float32}[]
for i in 1:19
    push!(env_covariates,
        SDMLayer(
            chelsa_provider;
            layer = "BIO$i",
            SDT.boundingbox(switzerland)...
        )
    )
end
```

```
env_covariates = map(
    layername -> Float32.(SDMLayer(
        chelsa_provider;
        layer = layername,
        SDT.boundingbox(switzerland)...
    )),
    layers(chelsa_provider)
);
```

Now we can visualize the first layer, and our polygon. We'll plot the first environmental layer with `heatmap`, and we'll pass `color=:white` and `linewidth=3` to make our polygon easier to see.

```
heatmap(env_covariates[begin])
lines!(switzerland, color=:white, linewidth=3)
current_figure()
```


47.5
47.0
46.5
46.0

Note that although our raster has the same extent as our polygon, it extends outside our polygon's border. We can fix this with the `mask!` method.

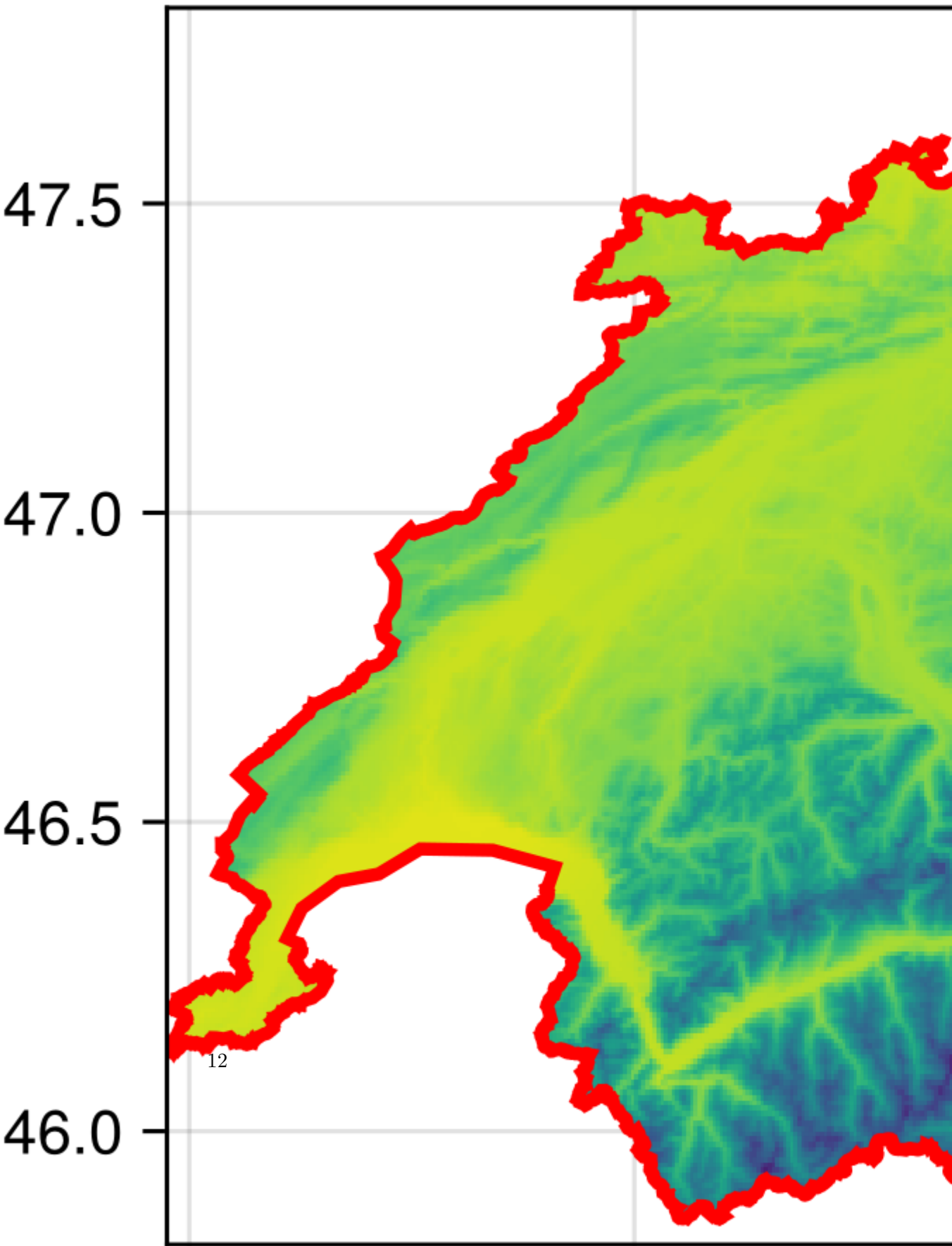
Masking the environmental layers

```
mask!(env_covariates, switzerland)
```

```
19-element Vector{SDMLayer{Float32}}:
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
```

and we can verify this worked by plotting it again:

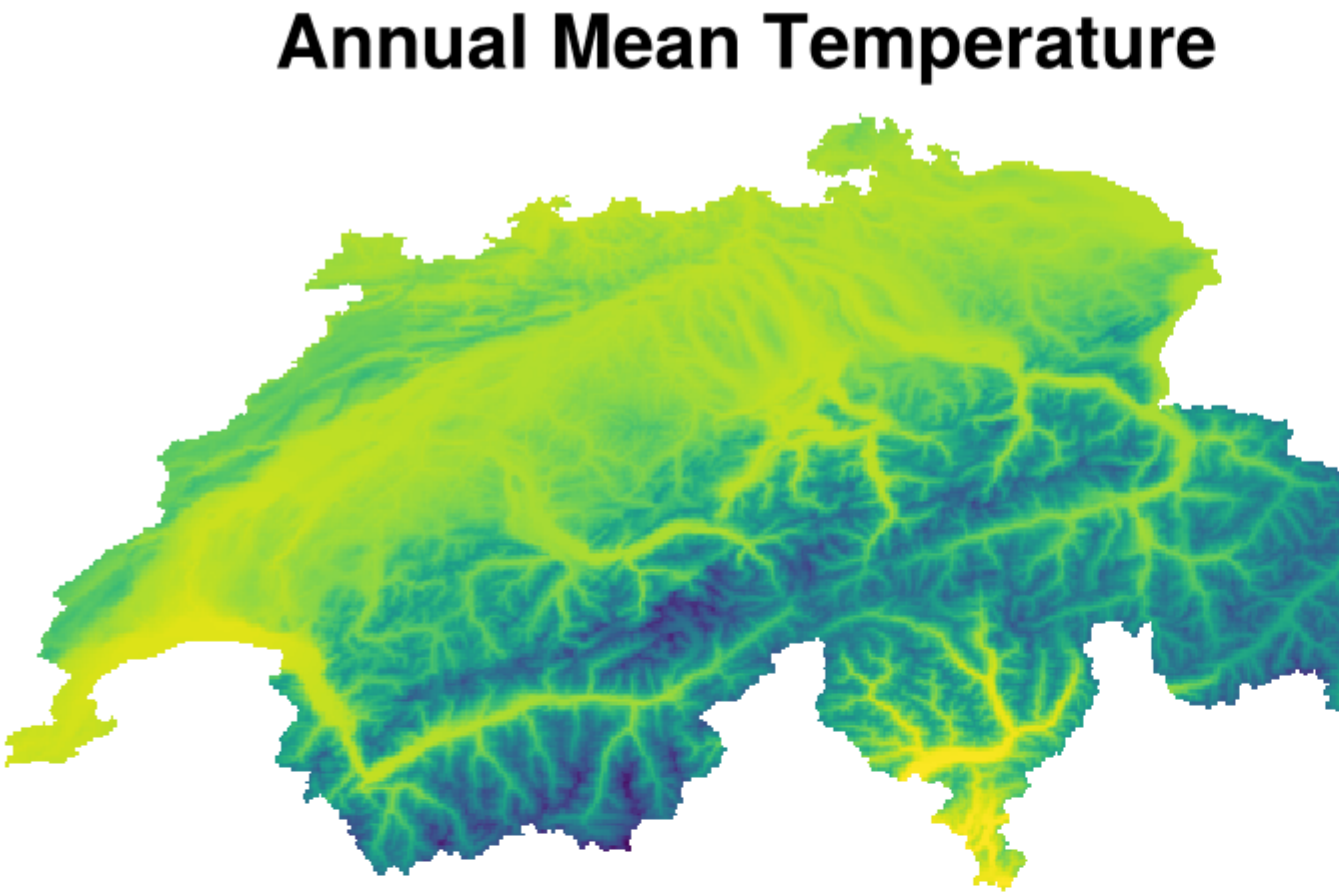
```
heatmap(env_covariates[begin])
lines!(switzerland, color=:red, linewidth=3)
current_figure()
```

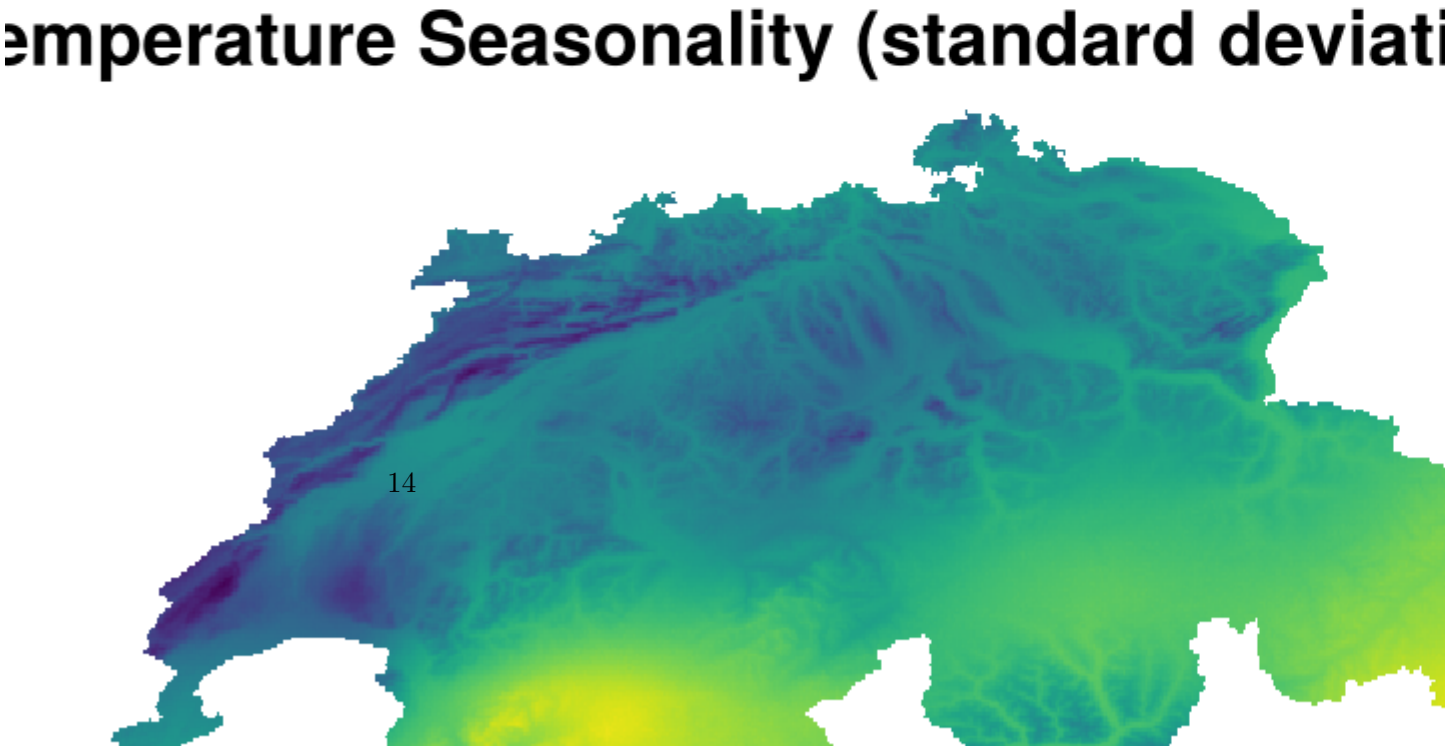

47.5
47.0
46.5
46.0

Let's plot a few of them.

Code for plotting multiple layers

```
layers_to_plot = [1, 4, 12, 15]

f = Figure()
for (i,ci) in enumerate(CartesianIndices((1:2,1:2)))
    this_layer = layers(chelsa_provider)[layers_to_plot[i]]
    ax = Axis(
        f[ci[1], ci[2]],
        title=layerdescriptions(chelsa_provider)[this_layer],
        titlesize=12
    )
    heatmap!(ax, env_covariates[layers_to_plot[i]])
    hidedecorations!(ax)
    hidespines!(ax)
end
```


Annual Mean Temperature


emperature Seasonality (standard deviati

Downloading occurrence data

Now we'll use the **GBIF.jl** subpackage to download occurrence data from the Global Biodiversity Information Facility (GBIF) for our species, *Turdus torquatus*. We do this with the `taxon` method.

```
ouzel = taxon("Turdus torquatus")
```

```
GBIF taxon -- Turdus torquatus
```

We'll then use the `occurrences` method to setup a data download from GBIF. We pass the taxon, `ouzel`, and an environmental covariate (with `first(env_covariates)`) representing the extent from which we want to download occurrences. The function also takes keyword arguments that are specified in the GBIF API.

```
presences = occurrences(
    ouzel,
    first(env_covariates),
    "occurrenceStatus" => "PRESENT",
    "limit" => 300,
    "country" => "CH",
    "datasetKey" => "4fa7b334-ce0d-4e88-aaae-2e0c138d049e",
)
```

```
GBIF records: downloaded 300 out of 1404
```

Note that this only downloads the first 300 occurrences, because the total number of records can vary drastically depending on the species and extent, and the GBIF streaming API has a hard limit at 200000 records, and querying large amounts of using the streaming API is woefully inefficient. For data volumes above 10000 observations, the suggested solution is to rely on the download interface on GBIF.

We can use the `count` method to determine how many total records match our criteria

```
count(presences)
```

```
1404
```

Because this is a reasonable number, we can download the rest of the occurrences using a while loop, and a the `occurrences!` method to iterate and download the remaining occurrences.

```
while length(presences) < count(presences)
    occurrences!(presences)
end
```

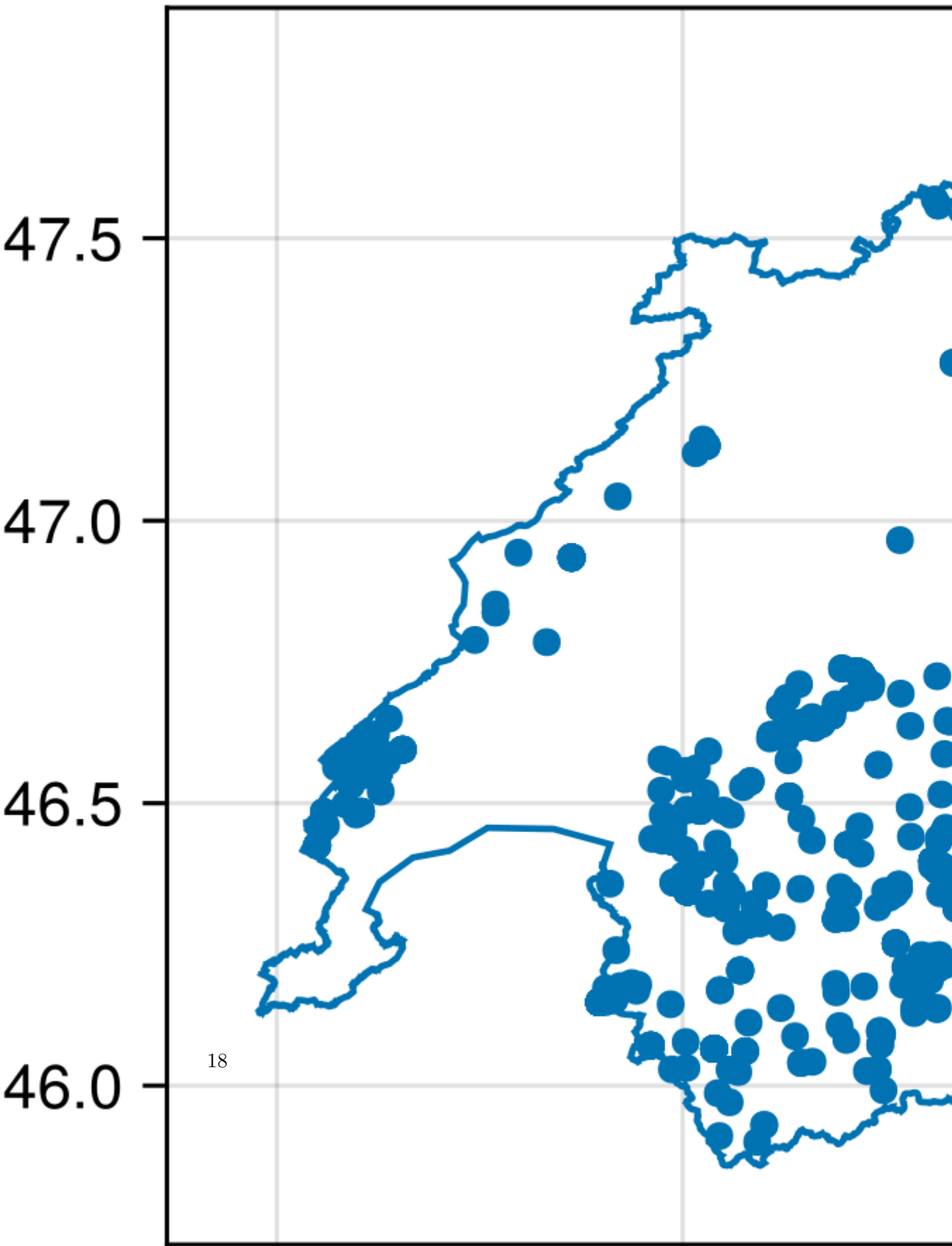
GBIF has a built-in Table.jl API, which means we can easily convert the occurrence records to a `DataFrame`:

```
DataFrame(presences)
```

	key	datasetKey	dataset	publishingOrgKey	
	Int64	String	Missing	String	
1	4709728505	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
2	4631294555	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
3	4717153698	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
4	4703405324	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
5	4698992061	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
6	4611055512	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
7	4794340181	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
8	4639401120	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
9	4779612275	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
10	4723809355	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
11	4724185634	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
12	4799483908	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
13	4798123478	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
14	4839563162	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
15	4716845682	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
16	4812470451	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
17	4726461633	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
18	4714731358	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
19	4804323453	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
20	4664223632	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
21	4638603293	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
22	4726706419	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
23	4613708697	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
24	4614664263	4fa7b334-ce0d-4e88-aaae-2e0c138d049e	*missing*	e2e717bf-551a-4917-bdc9-4fa0f342c530	..
...	...	...	...	...	

However, SDeMo is designed to work with the result from GBIF directly. For example, we can plot them with `scatter!`

```
lines(switzerland)
scatter!(presences)
current_figure()
```


47.5
47.0
46.5
46.0

We can also convert the occurrences into a raster with `true` values at the location of occurrences using the `mask` function. This will be useful for us in the next section.

```
presencelayer = mask(first(env_covariates), presences)
```

```
A 240 × 546 layer with 70065 Bool cells
Projection: +proj=longlat +datum=WGS84 +no_defs
```

Computing statistics with occurrences

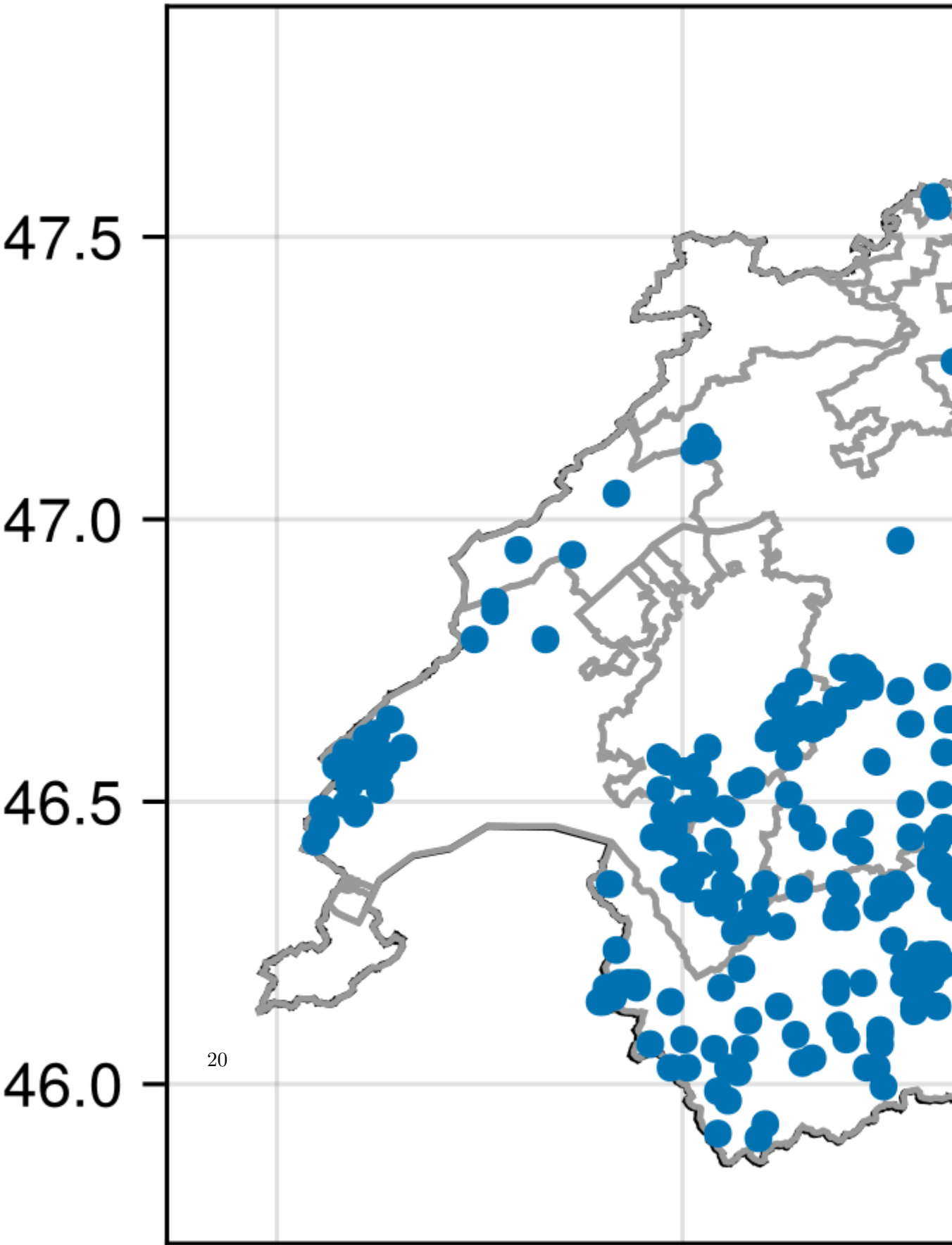
Here, we'll show how we can work with occurrence data and polygons. First, we'll download data on the Swiss cantons (states), using the GADM polygon database.

```
gadm_provider = PolygonData(GADM, Countries)
swiss_states = getpolygon(gadm_provider; country="CHE", level=1)
```

```
FeatureCollection with 26 features, each with 2 properties
```

Next we'll plot each state along with presence records.

```
lines(switzerland, color=:black)
lines!(swiss_states, color=:grey60)
scatter!(presencelayer)
current_figure()
```


47.5
47.0
46.5
46.0

Next we'll use the `byzone` method to compute the total number of presences in each state. We pass `sum` as the method to apply to each region, and `presencelayer` as the layer to apply `sum` to.

```
pres_per_state = Dict(
    byzone(sum, presencelayer, [x for x in swiss_states], [x.properties["Name"] for x in swiss
)
```

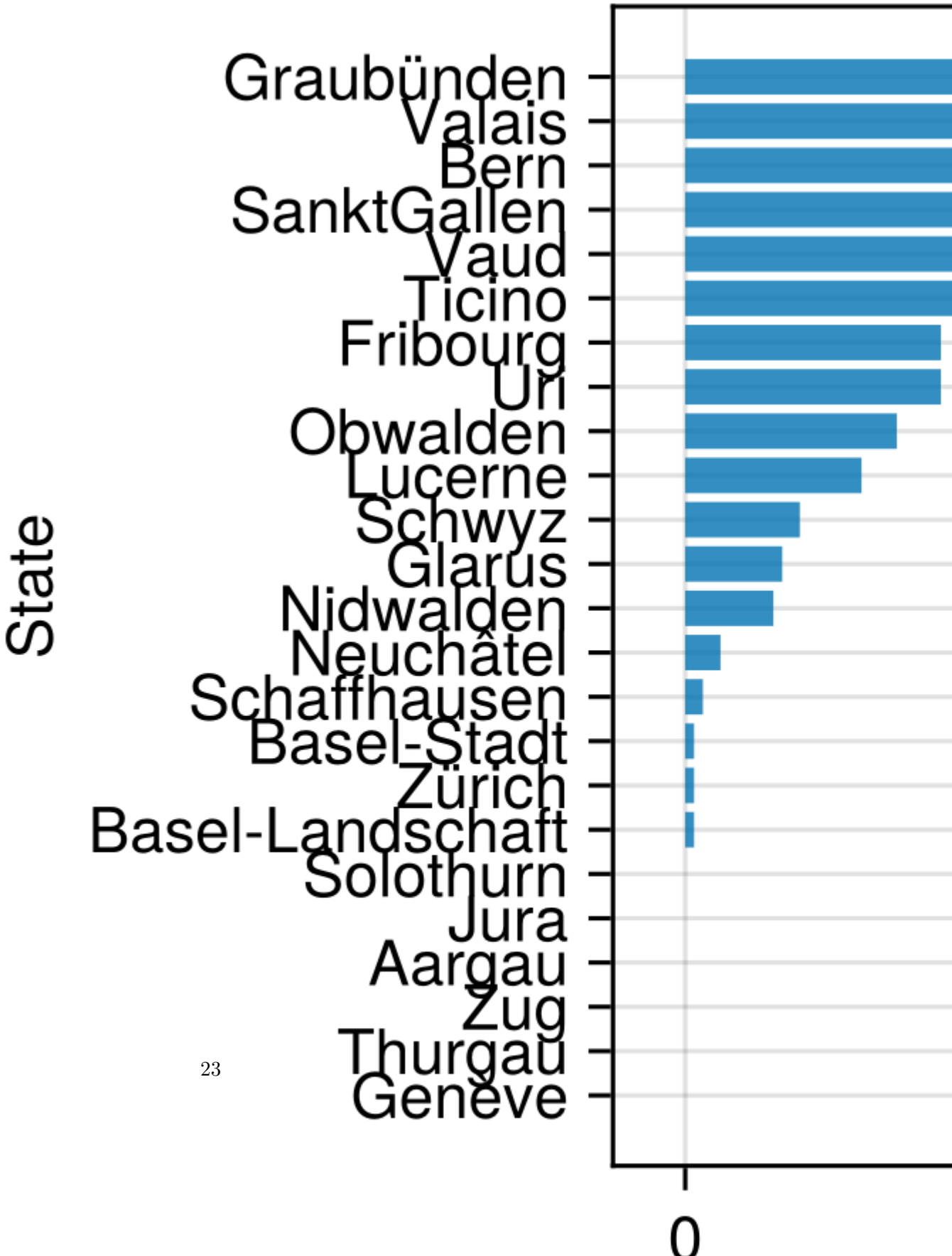
```
Dict{String, Int64} with 24 entries:
  "Genève"           => 0
  "Uri"              => 29
  "Ticino"           => 37
  "Thurgau"          => 0
  "Zug"              => 0
  "Schwyz"           => 13
  "Lucerne"          => 20
  "Obwalden"         => 24
  "Vaud"             => 56
  "Nidwalden"        => 10
  "SanktGallen"      => 68
  "Graubünden"       => 164
  "Neuchâtel"        => 4
  "Bern"             => 93
  "Fribourg"         => 29
  "Basel-Landschaft" => 1
  "Aargau"           => 0
  "Zürich"           => 1
  "Valais"           => 134
                     =>
```

Finally, we'll plot the total number of occurrences in each state as a bar chart.

Code for plotting presences by state

```
presence_cts = collect(values(pres_per_state))
sort_idx = sortperm(presence_cts)
state_names = collect(keys(pres_per_state))


f = Figure()
ax = Axis(
    f[1,1],
    xlabel = "Number of Occurrences",
    ylabel = "State",
    yticks=(1:length(state_names), state_names[sort_idx])
)
barplot!(ax, presence_cts[sort_idx], direction=:x)
```

```
Plot{barplot, Tuple{Vector{Point{2, Float64}}}}
```


State
Graubünden
Valais
Bern
SanktGallen
Vaud
Ticino
Fribourg
Uri
Obwalden
Lucerne
Schwyz
Glarus
Nidwalden
Neuchâtel
Schaffhausen
Basel-Stadt
Zürich
Basel-Landschaft
Solothurn
Jura
Aargau
Zug
Thurgau
Genève
0

Associating Environmental Covariates with Occurrences

Next, we'll show how we associate the data in our environmental covariates with each occurrence point. First, let's select the environmental covariates that represent mean annual temperature (BIO1), and annual precipitation (BIO12).

```
temperature, precipitation = env_covariates[[1,12]]
```

```
2-element Vector{SDMLayer{Float32}}:
    A 240 × 546 layer (70065 Float32 cells)
    A 240 × 546 layer (70065 Float32 cells)
```

We can simply index the layers by the `presences` object to select the value of the covariate at each location.

```
temp, precip = temperature[presences], precipitation[presences]
```

```
(Float32[2779.0, 2767.0, 2741.0, 2741.0, 2775.0, 2760.0, 2798.0, 2731.0, 2795.0, 2828.0  …  273
```

Note that CHELSA doesn't provide data in commonly used units — the transformations to convert them into typical units can be found in their [documentation]. The relevant transformations for temperature and precipitation are applied below.

```
temp = 0.1temp .- 271
precip = 0.1precip
```

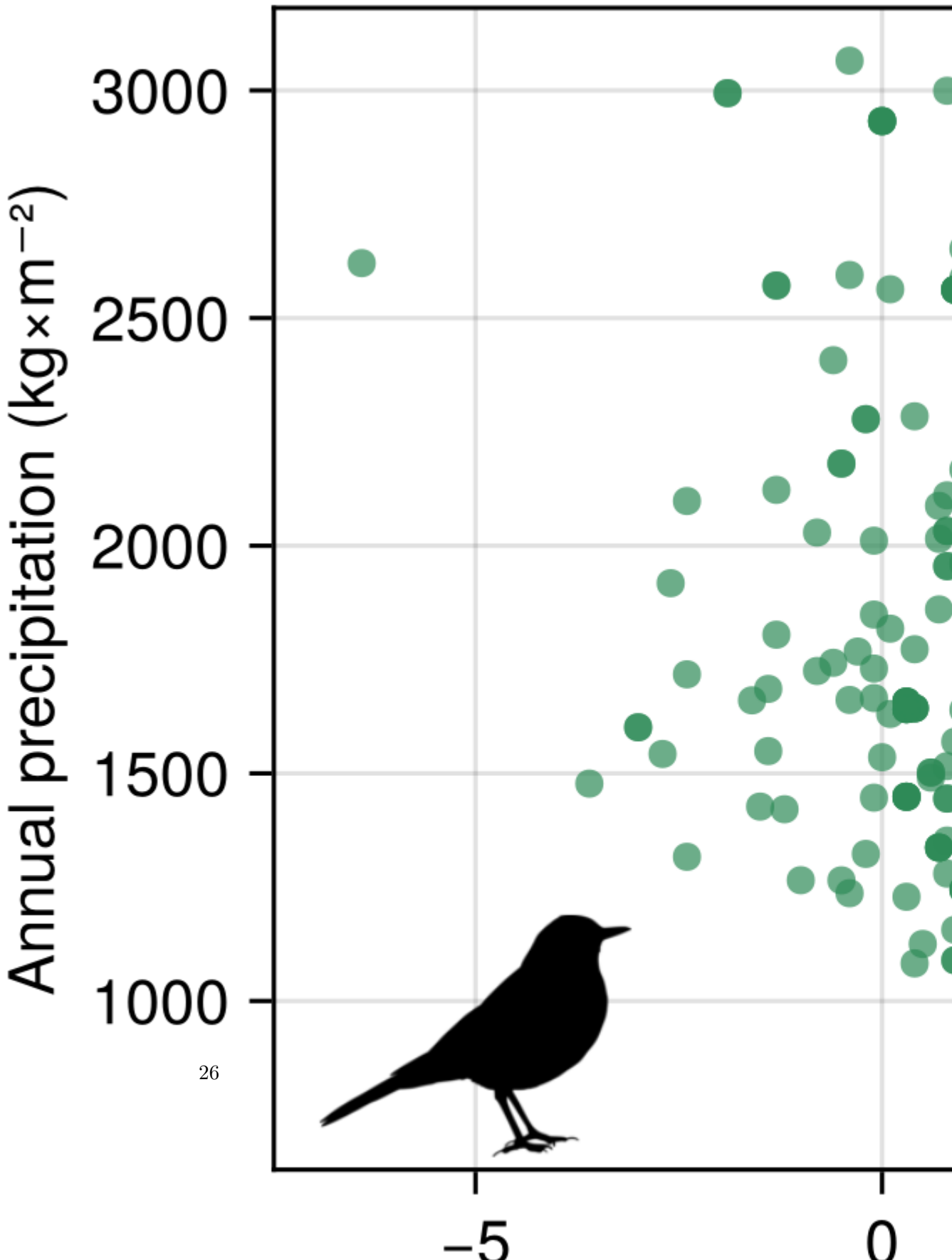
Finally, we can add a silhoutte of our taxon to the plot by downloading it using the Phylopic subpackage.

```
taxon_silhoeutte = Phylopic.imagesof(ouzel)
```

```
PhylopicSilhouette("Turdus migratorius", Base.UUID("7be55586-e993-4eb3-88a0-9ec7a2ba0c7d"))
```

This can all be visualized using the code below, which shows a clear negative correlation between temperature and precipitation at the occurrence locations.

```
f = Figure()
ax = Axis(
    f[1,1],
    xlabel="Annual mean temperature (°C)",
    ylabel="Annual precipitation (kg×m ²)"
)
scatter!(ax, 0.1temperature[presences].-271, 0.1precipitation[presences], color=(:seagreen4, 0
silhouetteplot!(ax, -5., 1000.0, taxon_silhoeutte; markersize=70)
f
```


3000
2500
2000
1500
1000
Annual precipitation (kg×m⁻²)
−5
0

Building a Simple Species Distribution Model

Now that we have obtained both occurrence and environmental data, and explored it, we are ready to fit a species distribution model.

The **SDeMo** subpackage provides methods for data preparation, training, validation, and interpretation, as well as several built-in models. Before we fit a more complicated machine-learning model, we'll start by beocming familiar with the **SDeMo** API using a simpler model, the *Naive-Bayes classifier*.

Sampling Pseudo-Absences

All of the models we use for binary classification, including *Naive-Bayes*, require presence-absence data. However, for the vast majority of species, we don't have records of true species absences because these typically expensive monitoring programs, in contrast to the widespread availability of presence data on GBIF, which is largely crowdsourced from community science platforms like *iNaturalist*.

To deal with this, a widespread method is generating *pseudo*-absences, which rely on hueristics to select sites where it is *very unlikely* that the target species is present. There is a deep literature on methods to select the locations and number of pseudoabsences (**tk cites**). Here we will use a method called *background thickening*, which means the probability a given location is marked as a pseudoabsence grows with the *minimum* distance to nearest presence.

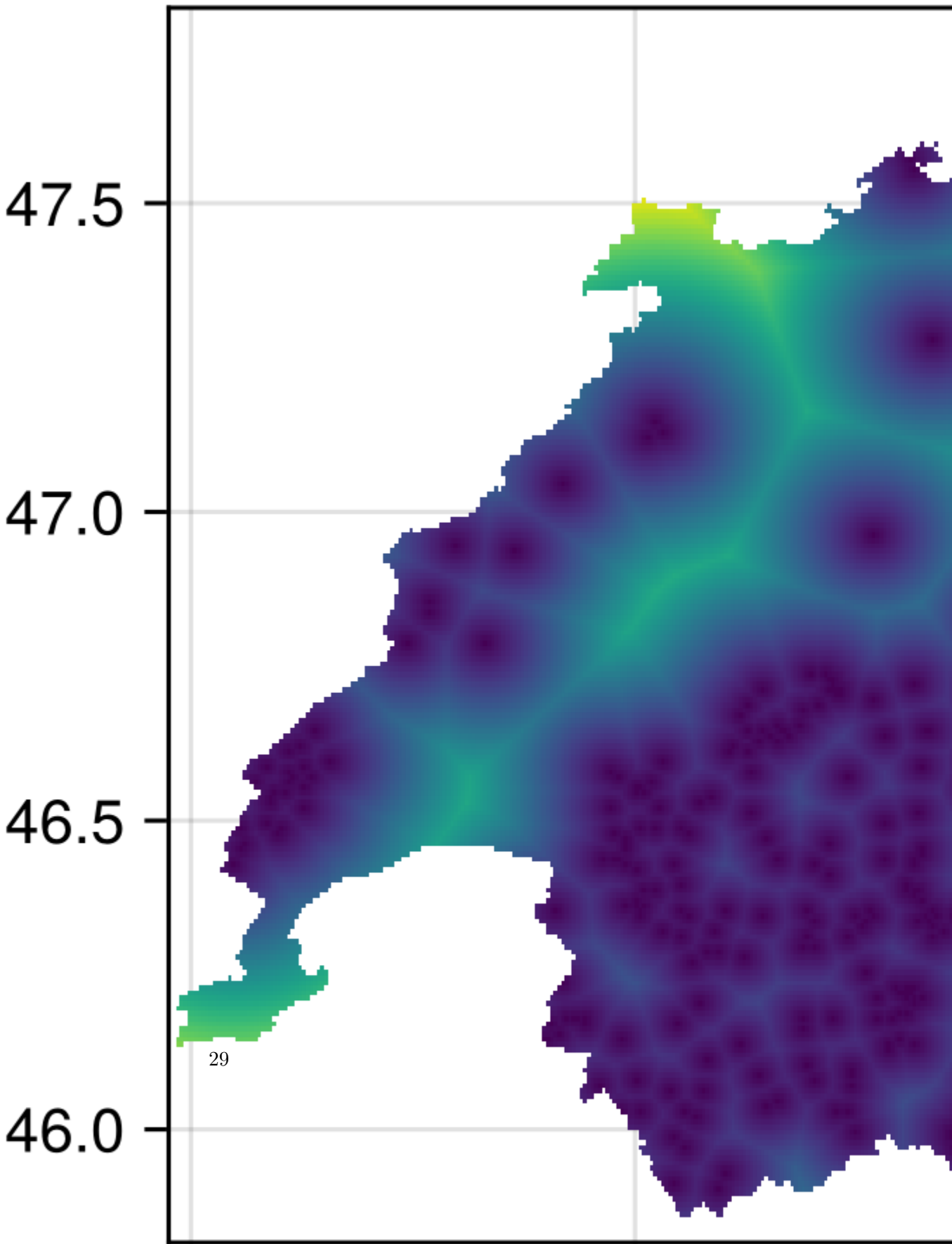
We can implement background thickening this using the `pseudoabsencemask` method, first with the `DistanceToEvent` technique, which generates a layer where each value is the distance (in kilometers) to the nearest presence record.

```
background = pseudoabsencemask(DistanceToEvent, presencelayer)
```

```
A 240 × 546 layer with 70065 Float64 cells
 Projection: +proj=longlat +datum=WGS84 +no_defs
```

We can then visualize this using `heatmap`.

```
heatmap(background)
```


47.5
47.0
46.5
46.0

We *could* draw pseudoabsences from this, but it is also typically a good idea to add a *buffer* around each presence, which are not allowed to include pseudoabsences. The justification for this is it's unlikely to be truly absent in locations very close to an observed presence. Here, we'll use a buffer distance of 4 kilometers, and mask those regions out using the `nodata` method.

```
buffer_distance = 4 # In kilometers
buffered_background = nodata(background, d -> d < buffer_distance)
```

```
A 240 × 546 layer with 45448 Float64 cells
 Projection: +proj=longlat +datum=WGS84 +no_defs
```

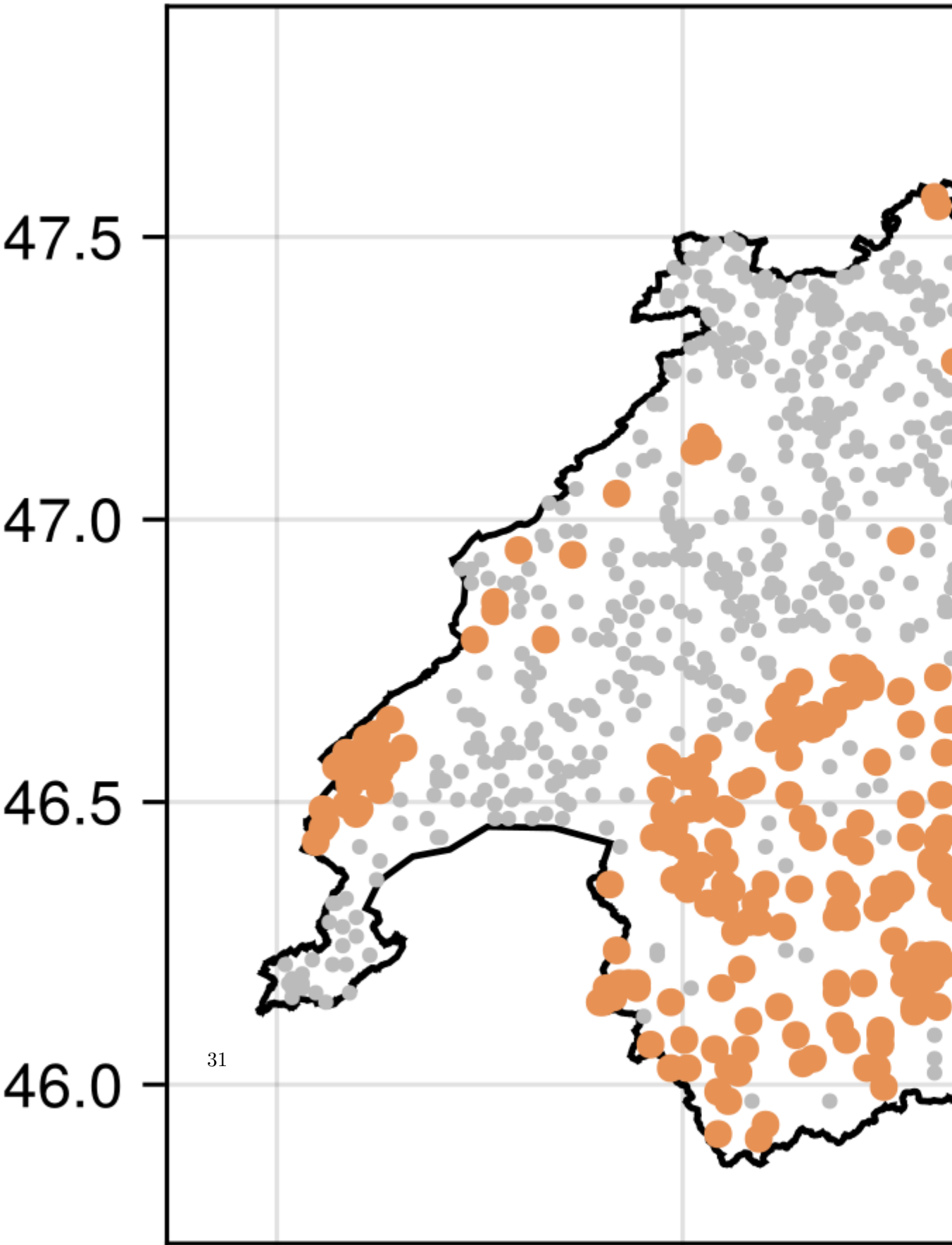
Finally, we'll sample pseudoabsences using the `backgroundpoints` method. We'll choose to sample twice as many pseudoabsences as there are presences. In real workflows, it' important to determine the sensitivity of a model to the number of pseudoabsences, but given this tutorial is focused on interpretable machine learning, we'll stick with number of pseudoabsences for each example.

```
num_pseudoabsences = 2sum(presencelayer)
pseudoabsences = backgroundpoints(buffered_background, num_pseudoabsences)
```

```
A 240 × 546 layer with 45448 Bool cells
 Projection: +proj=longlat +datum=WGS84 +no_defs
```

Finally, we can visualize the pseudoabsences using `scatter!`, just as we would for presences. Below, presences are in orange, and pseudoabsences are in grey.

```
lines(switzerland, color=:black)
scatter!(presencelayer, color=colorant"#e79154")
scatter!(pseudoabsences, color=colorant"#bbb", markersize=5)
current_figure()
```

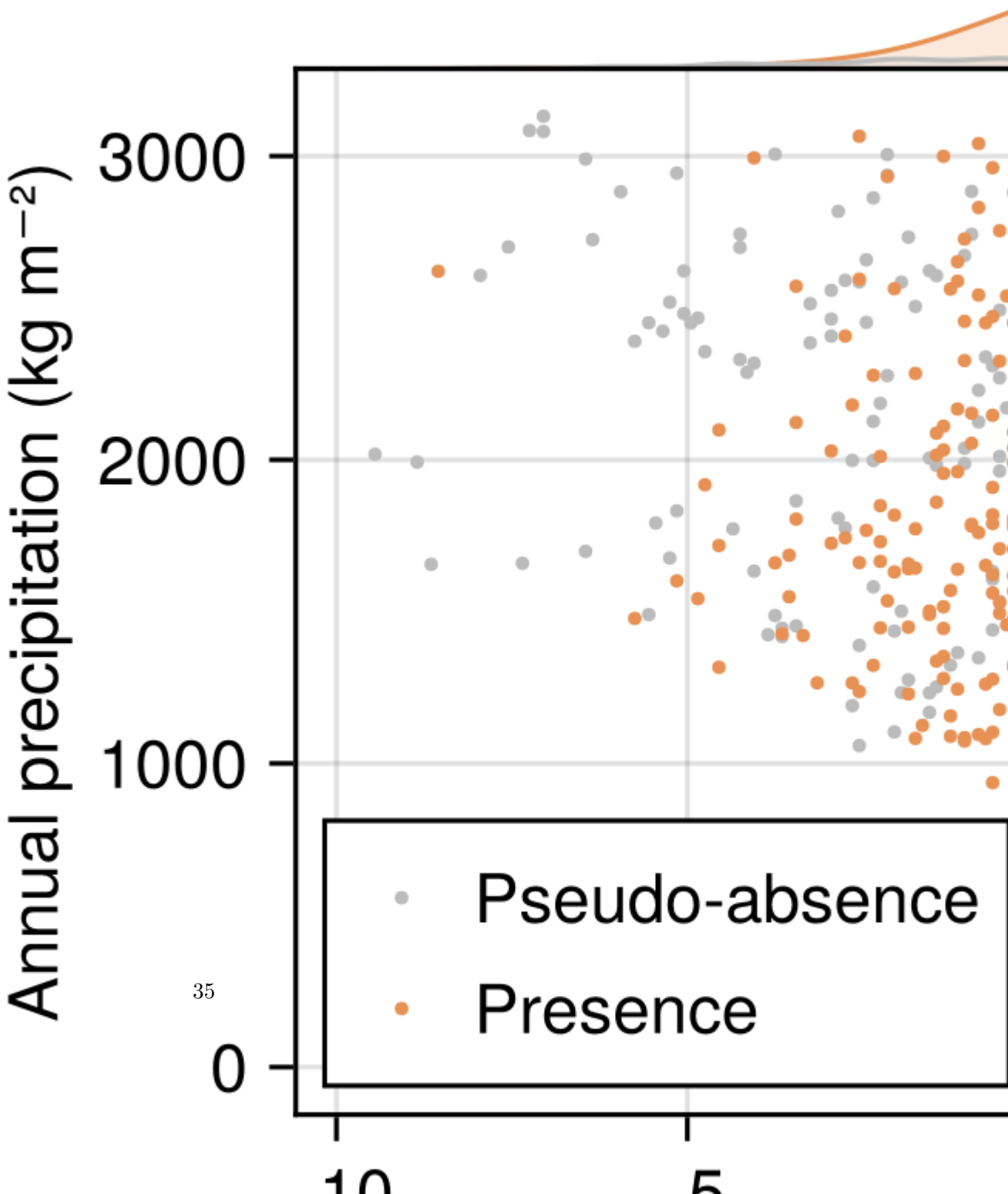

47.5
47.0
46.5
46.0

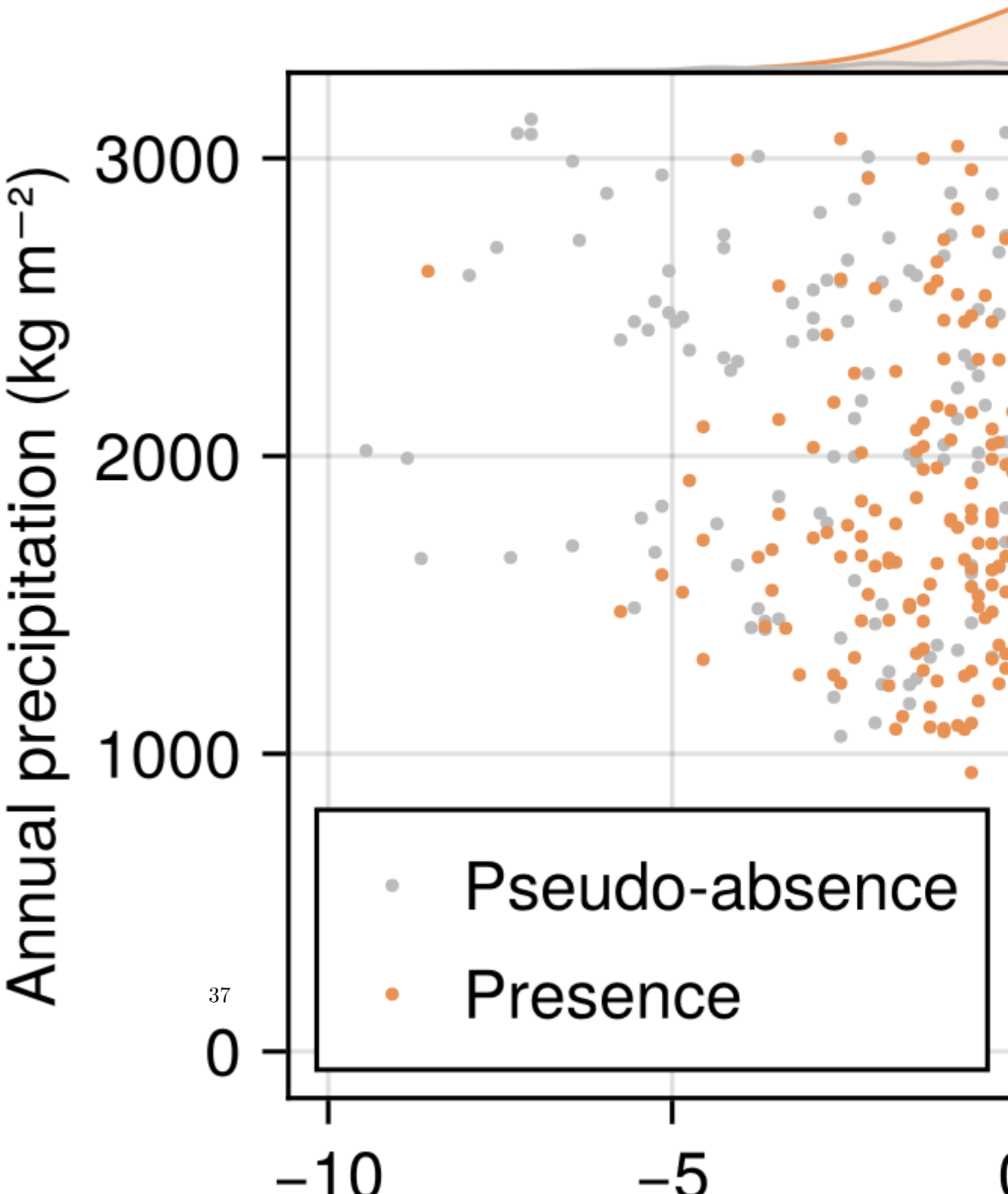
We can see there is a clear geographic distinction in the regions where presences and absences are, but crucially we need them to be if different regions of environmental space.

We visualize the density of presences and absences in environmental space below.

Code for plotting density in environmental space

```
abcol = colorant"#bbb"
prcol = colorant"#e79154"

_range = (
    absent = abcol,
    present = prcol,
    absentbg = (abcol, 0.2),
    presentbg = (prcol, 0.2),
)
bkcol = (
    nodata = colorant"#DDDDDD",
    generic = colorant"#222222",
    sdm = _range,
)

temp_idx, precip_idx = 1, 12

tmp, precip = 0.1env_covariates[temp_idx] - 273.15, 0.1env_covariates[precip_idx]

temp_pres = tmp.grid[presencelayer.grid]
temp_abs = tmp.grid[pseudoabsences.grid]

precip_pres = precip.grid[presencelayer.grid]
precip_abs = precip.grid[pseudoabsences.grid]

f = Figure()

gl = f[1,1] = GridLayout()

axtemp = Axis(gl[1,1])
density!(axtemp, temp_pres, color=bkcol.sdm.presentbg, strokecolor=bkcol.sdm.present, strokew
density!(axtemp, temp_abs, color=bkcol.sdm.absentbg, strokecolor=bkcol.sdm.absent, strokewidt

axprec = Axis(gl[2,2])
density!(axprec, precip_pres, color=bkcol.sdm.presentbg, strokecolor=bkcol.sdm.present, strok
density!(axprec, precip_abs, color=bkcol.sdm.absentbg, strokecolor=bkcol.sdm.absent, strokewi

axboth = Axis(gl[2,1], xlabel="Mean air temperature (°C)", ylabel = "Annual precipitation (kg
scatter!(axboth, temp_abs, precip_abs, color=bkcol.sdm.absent, markersize=4, label="Pseudo-ab
scatter!(axboth, temp_pres, precip_pres, color=bkcol.sdm.present, markersize=4, label="Preser

axislegend(position = :lb)

hidespines!(axtemp, :l,34r, :t)
hidespines!(axprec, :b, :r, :t)
hidedecorations!(axtemp, grid = true)
hidedecorations!(axprec, grid = true)
ylims!(axtemp, low = 0)
xlims!(axprec, low = 0)
colgap!(gl, 0)
rowgap!(gl, 0)
```


Annual precipitation (kg m⁻²)
3000
2000
1000
0
Pseudo-absence
Presence
10
5


Annual precipitation (kg m⁻²)
3000
2000
1000
0
−10
−5
Pseudo-absence
Presence

Training a Simple SDM

Now that we have presences and pseudoabsences, we are finally ready to train our model.**SDeMo** uses a single `SDM` type that chains together the data transformation, the model, and data to fit on.

We'll build a *Naive-Bayes* classifier that first applies Principal Component Analysis (PCA) *only to the training data*[5].

[5] It's crucial that PCA is only applied to the training data to avoid introducing *data leakage*.

```
nb_sdm = SDM(PCATransform, NaiveBayes, env_covariates, presencelayer, pseudoabsences)
```

```
SDeMo.PCATransform → SDeMo.NaiveBayes → P(x)   0.5
```

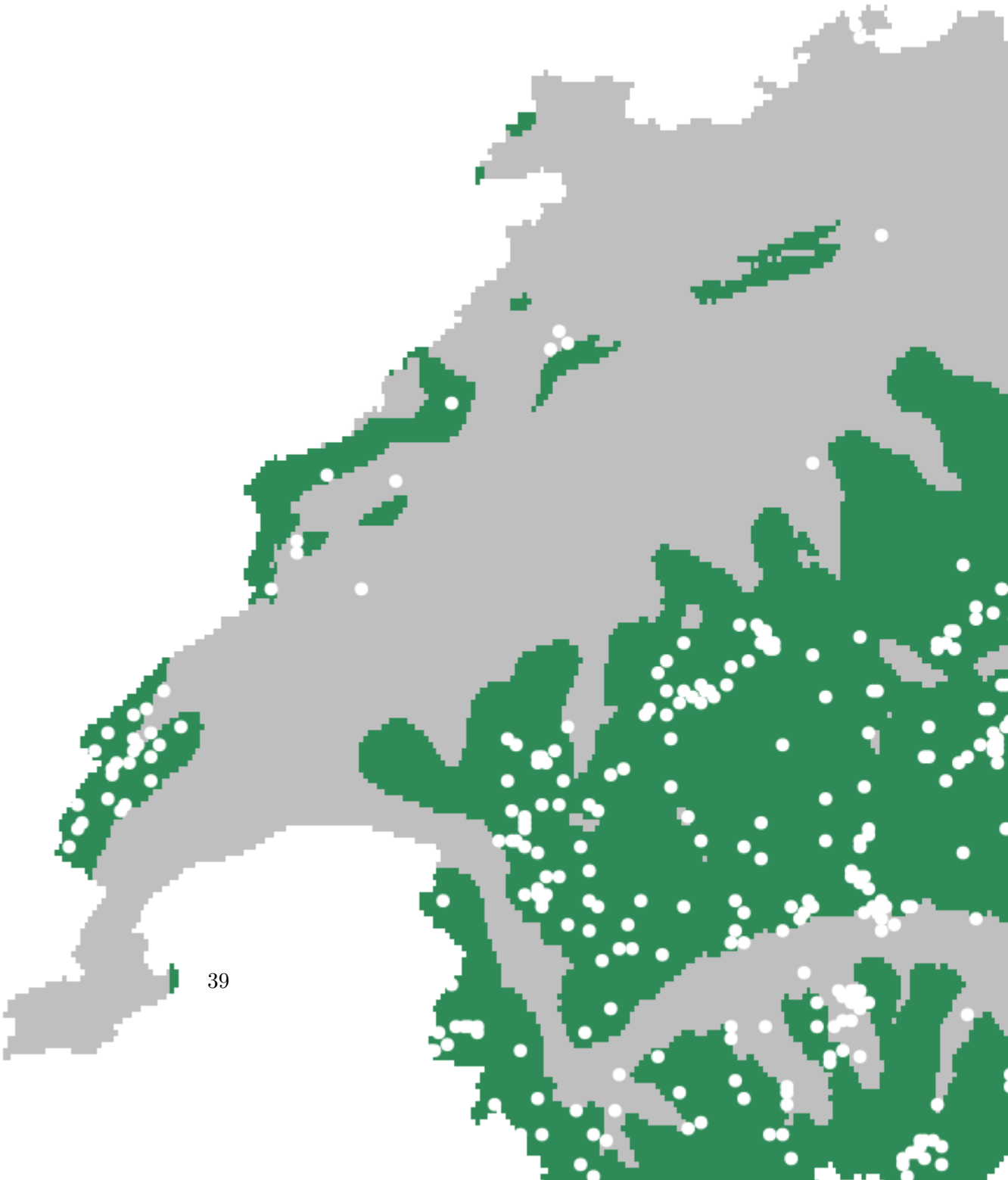
```
train!(nb_sdm)
```

```
SDeMo.PCATransform → SDeMo.NaiveBayes → P(x)   0.187
```

```
prd = predict(nb_sdm, env_covariates; threshold = true)
```

```
  A 240 × 546 layer with 70065 Bool cells
   Projection: +proj=longlat +datum=WGS84 +no_defs
```

```
f = Figure()
ax = Axis(f[1,1])
heatmap!(ax, prd, colormap=[:grey75, :seagreen4])
scatter!(ax, presencelayer, color=:white, markersize=4)
hidedecorations!(ax)
hidespines!(ax)
f
```



Computing ROC-AUC and PR-AUC

```
folds = kfold(nb_sdm)
cv = crossvalidate(nb_sdm, folds)
```

```
(validation = ConfusionMatrix[[TP: 68, TN 88, FP 50, FN 3], [TP: 64, TN 90, FP 46, FN 6], [TP:
```

```
measures = [mcc, ppv, npv, plr, nlr]
cvresult = [measure(set) for measure in measures, set in cv]
nullresult = [measure(null(nb_sdm)) for measure in measures, null in [coinflip, noskill]]
pretty_table(
    hcat(string.(measures), hcat(cvresult, nullresult));
    alignment = [:l, :c, :c, :c, :c],
    #backend = Val(:markdown),
    header = ["Measure", "Validation", "Training", "Coin-flip", "No-skill"],
    formatters = ft_printf("%5.3f", [2, 3, 4, 5]),
)
```

Measure	Validation	Training	Coin-flip	No-skill
mcc	0.505	0.510	-0.323	-0.000
ppv	0.547	0.549	0.338	0.338
npv	0.936	0.938	0.338	0.662
plr	2.378	2.381	0.512	1.000
nlr	0.136	0.129	1.954	1.000

```
folds = kfold(nb_sdm)
thresholds = LinRange(0.01, 0.99, 50)
cv = [crossvalidate(nb_sdm, folds; thr = t) for t in thresholds];
```

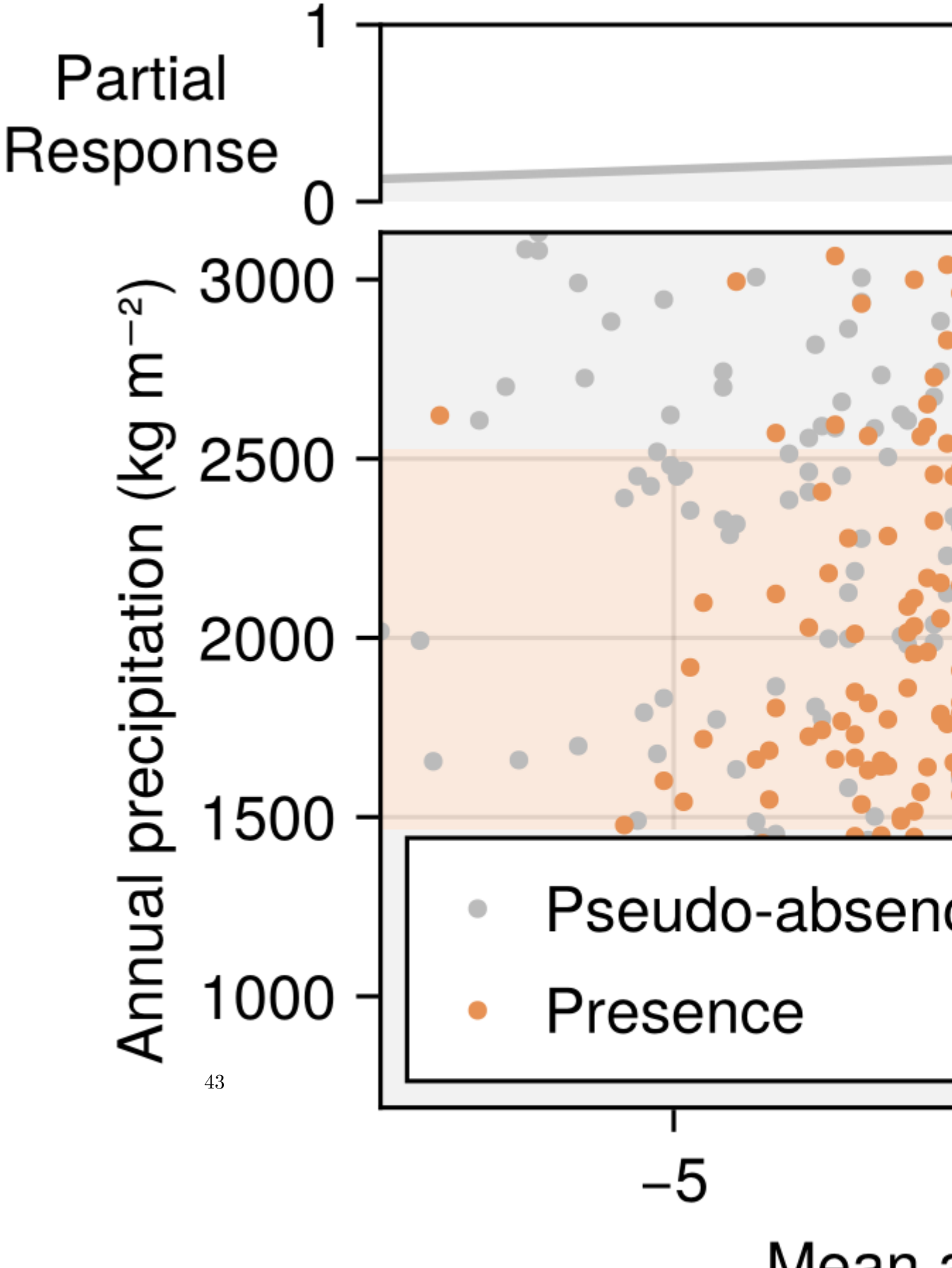
```
# TODO: make these functions
SDT.SDeMo.auc([tpr(rep.validation) for rep in cv], ([tnr(rep.validation) for rep in cv]))
#SDT.SDeMo.auc([tpr(rep.validation) for rep in cv], [ppv(rep.validation) for rep in cv], )
```

```
0.8062856176441721
```

Simple Explanations: Partial Response

```
f = Figure()

gl = f[1,1] = GridLayout()

axtemp = Axis(
    gl[1,1],
    xgridvisible=false,
    ygridvisible=false,
    xticksvisible=false,
    xticklabelsvisible=false,
    yticks=0:1,
    ylabel = "Partial\nResponse",
    ylabelrotation=0
)

_, pr_p_thres = partialresponse(nb_sdm, variables(nb_sdm)[4], threshold=true)
  = findfirst(x->x==0, pr_p_thres)
pr_t_x, pr_t_y = partialresponse(nb_sdm, variables(nb_sdm)[4], threshold=false)

xlims!(axtemp, extrema(pr_t_x)...)
ylims!(axtemp, 0, 1)

lines!(axtemp, pr_t_x[1:], pr_t_y[1:], color=bkcol.sdm.present, linewidth=2)
band!(axtemp, pr_t_x[1:], [0. for _ in pr_t_y[1:]], pr_t_y[1:], color=bkcol.sdm.presentbg)

lines!(axtemp, pr_t_x[:end], pr_t_y[:end], color=bkcol.sdm.absent, linewidth=2)
band!(axtemp, pr_t_x[:end], [0. for _ in pr_t_y[:end]], pr_t_y[:end], color = bkcol.sdm.absen

axprec = Axis(
    gl[2,2],
    xgridvisible=false,
    ygridvisible=false,
    yticksvisible=false,
    yticklabelsvisible=false,
    xticks=0:1,
    xlabel="Partial\nResponse",
    topspinecolor=colorant"#999",
    leftspinecolor=colorant"#999",
```

```
    rightspinecolor=colorant"#999",
    bottomspinecolor=colorant"#999",
)

pr_p_y, pr_p_x = partialresponse(nb_sdm, variables(nb_sdm)[12], threshold=false)
lines!(axprec, pr_p_x, pr_p_y, linewidth=2)

xlims!(axprec, 0, 1)
ylims!(axprec, extrema(pr_p_y)...)


prx, pry, prz = partialresponse(nb_sdm, variables(nb_sdm)[[1,12]]..., (50, 100); threshold = t
prx = 0.1prx .- 273.15
pry = 0.1pry

axboth = Axis(gl[2,1], xlabel="Mean air temperature (°C)", ylabel = "Annual precipitation (kg m
limits!(axboth, extrema(prx)..., extrema(pry)...)
heatmap!(axboth, prx, pry, prz, colormap=[:grey95, (bkcol.sdm.present, 0.2)])
scatter!(axboth, temp_abs, precip_abs, color=bkcol.sdm.absent, markersize=6, label="Pseudo-abse
scatter!(axboth, temp_pres, precip_pres, color=bkcol.sdm.present, markersize=6, label="Presence

axislegend(position = :lb)

hidespines!(axtemp, :b)
hidespines!(axprec, :l)

colgap!(gl, 7)
rowgap!(gl, 7)

colsize!(gl, 1, Relative(5/6))
rowsize!(gl, 2, Relative(5/6))

f
```


Partial
Response
1
0
3000
2500
2000
1500
1000
Annual precipitation (kg m⁻²)
Pseudo-absenc
Presence
−5

A more complex SDM – Boosted Regression Trees from scratch(-ish)

TODO: What is a BRT explanation: - What is a decision tree? - Decision Tree + Bagging = Random Forest - Random Forest + Boosting = BRT

Fitting a Decision Tree

```
dt_sdm = SDM(ZScore, DecisionTree, env_covariates, presencelayer, pseudoabsences)
```

```
SDeMo.ZScore → SDeMo.DecisionTree → P(x)   0.5
```

```
@time variables!(dt_sdm, ForwardSelection, kfold(dt_sdm, k=2))
```

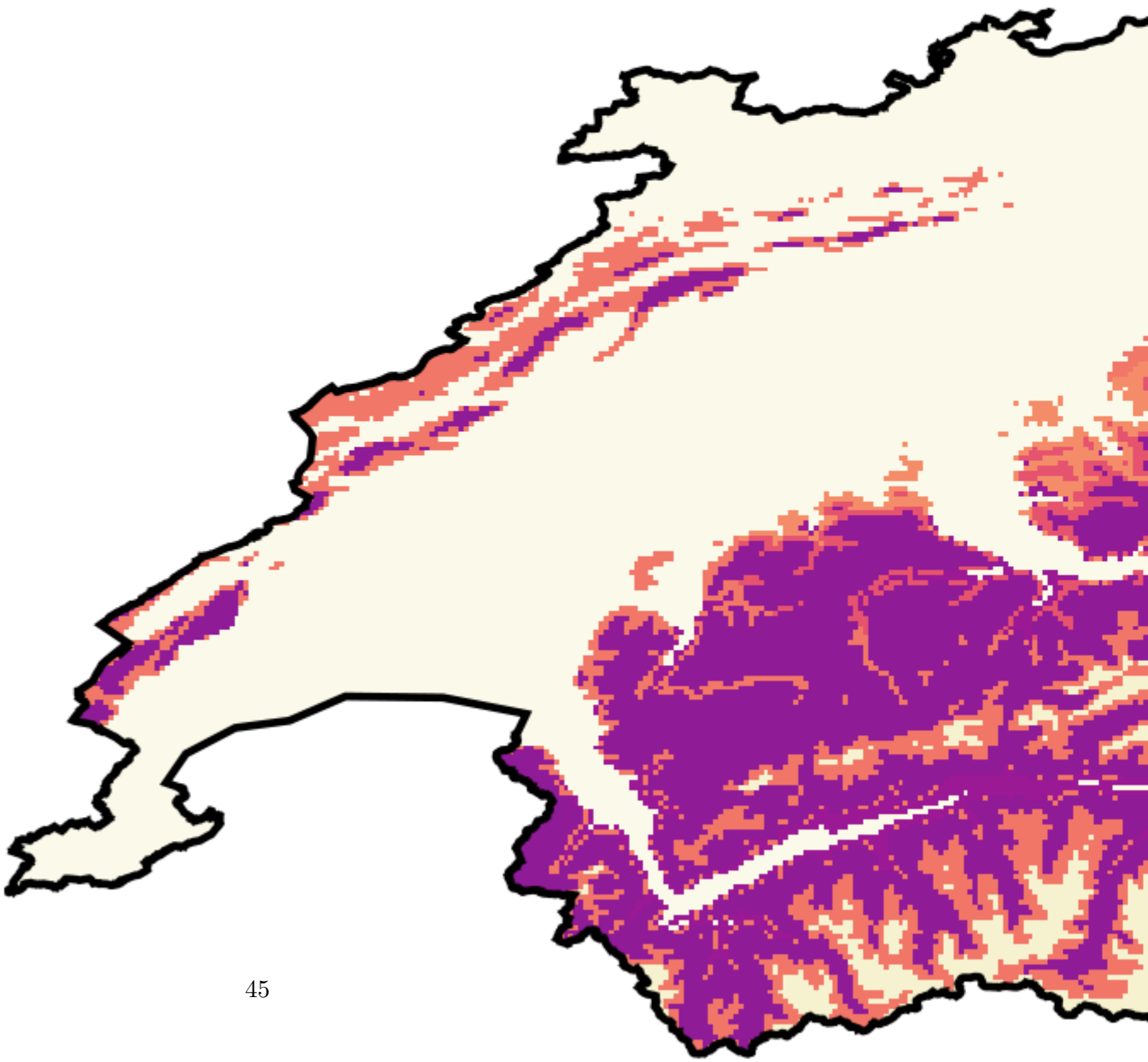
```
  2.649110 seconds (18.40 M allocations: 2.197 GiB, 6.35% gc time)
```

```
SDeMo.ZScore → SDeMo.DecisionTree → P(x)   0.586
```

```
prd = predict(dt_sdm, env_covariates; threshold = false)
```

```
   A 240 × 546 layer with 70065 Float64 cells
    Projection: +proj=longlat +datum=WGS84 +no_defs
```

```
f = Figure(; size = (600, 300))
ax = Axis(f[1, 1]; aspect = DataAspect(), title = "Prediction (decision tree)")
hm = heatmap!(ax, prd; colormap = :linear_worb_100_25_c53_n256, colorrange = (0, 1))
Colorbar(f[1, 2], hm)
lines!(ax, switzerland; color = :black)
hidedecorations!(ax)
hidespines!(ax)
f
```


Predic

TK compare DT to Naive Bayes

Making a random forest

```
single_dt_sdm = SDM(PCATransform, DecisionTree, env_covariates, presencelayer, pseudoabsences)
rf_sdm = Bagging(single_dt_sdm, 30)
```

```
{SDeMo.PCATransform → SDeMo.DecisionTree → P(x)   0.5} × 30
```

```
bagfeatures!(rf_sdm)
```

```
{SDeMo.PCATransform → SDeMo.DecisionTree → P(x)   0.5} × 30
```

```
#variables!(rf_sdm, AllVariables)
@time variables!(rf_sdm, ForwardSelection, kfold(rf_sdm, k=2); bagfeatures = true)
```

```
 49.986693 seconds (512.97 M allocations: 58.915 GiB, 14.22% gc time)
```

```
{SDeMo.PCATransform → SDeMo.DecisionTree → P(x)   0.303} × 30
```

```
train!(rf_sdm)
```

```
{SDeMo.PCATransform → SDeMo.DecisionTree → P(x)   0.303} × 30
```

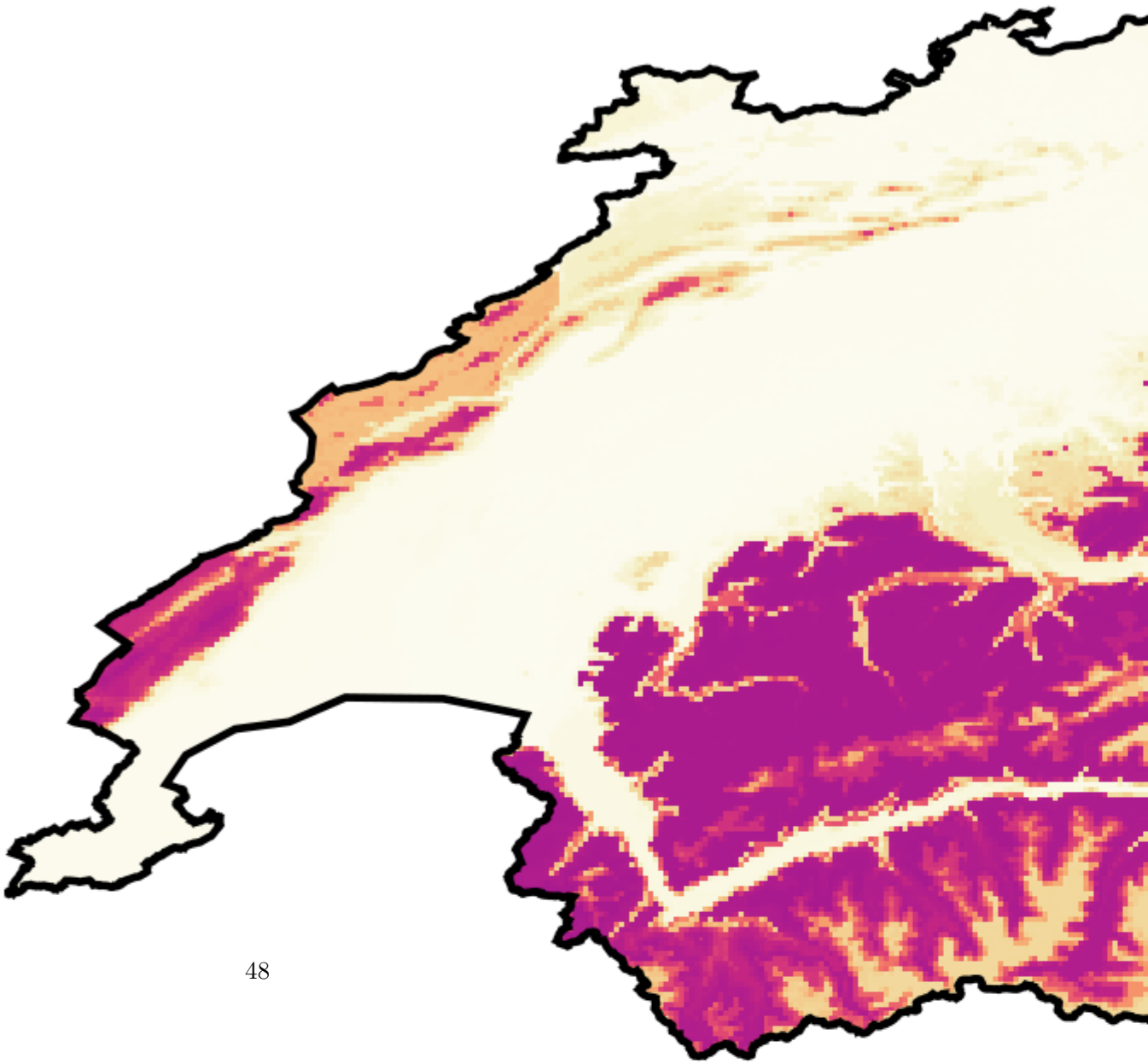
```
prd = predict(rf_sdm, env_covariates; threshold = false)
```

```
A 240 × 546 layer with 70065 Float64 cells
 Projection: +proj=longlat +datum=WGS84 +no_defs
```

```
unc = predict(rf_sdm, env_covariates; consensus = iqr, threshold = false)
```

```
A 240 × 546 layer with 70065 Float64 cells
 Projection: +proj=longlat +datum=WGS84 +no_defs
```

```
f = Figure(; size = (600, 600))
ax = Axis(f[1, 1]; aspect = DataAspect(), title = "Prediction")
hm = heatmap!(ax, prd; colormap = :linear_worb_100_25_c53_n256, colorrange = (0, 1))
Colorbar(f[1, 2], hm)
lines!(ax, switzerland; color = :black)
hidedecorations!(ax)
hidespines!(ax)
ax2 = Axis(f[2, 1]; aspect = DataAspect(), title = "Uncertainty")
hm =
    heatmap!(ax2, quantize(unc); colormap = :linear_gow_60_85_c27_n256, colorrange = (0, 1))
Colorbar(f[2, 2], hm)
lines!(ax2, switzerland; color = :black)
hidedecorations!(ax2)
hidespines!(ax2)
f
```



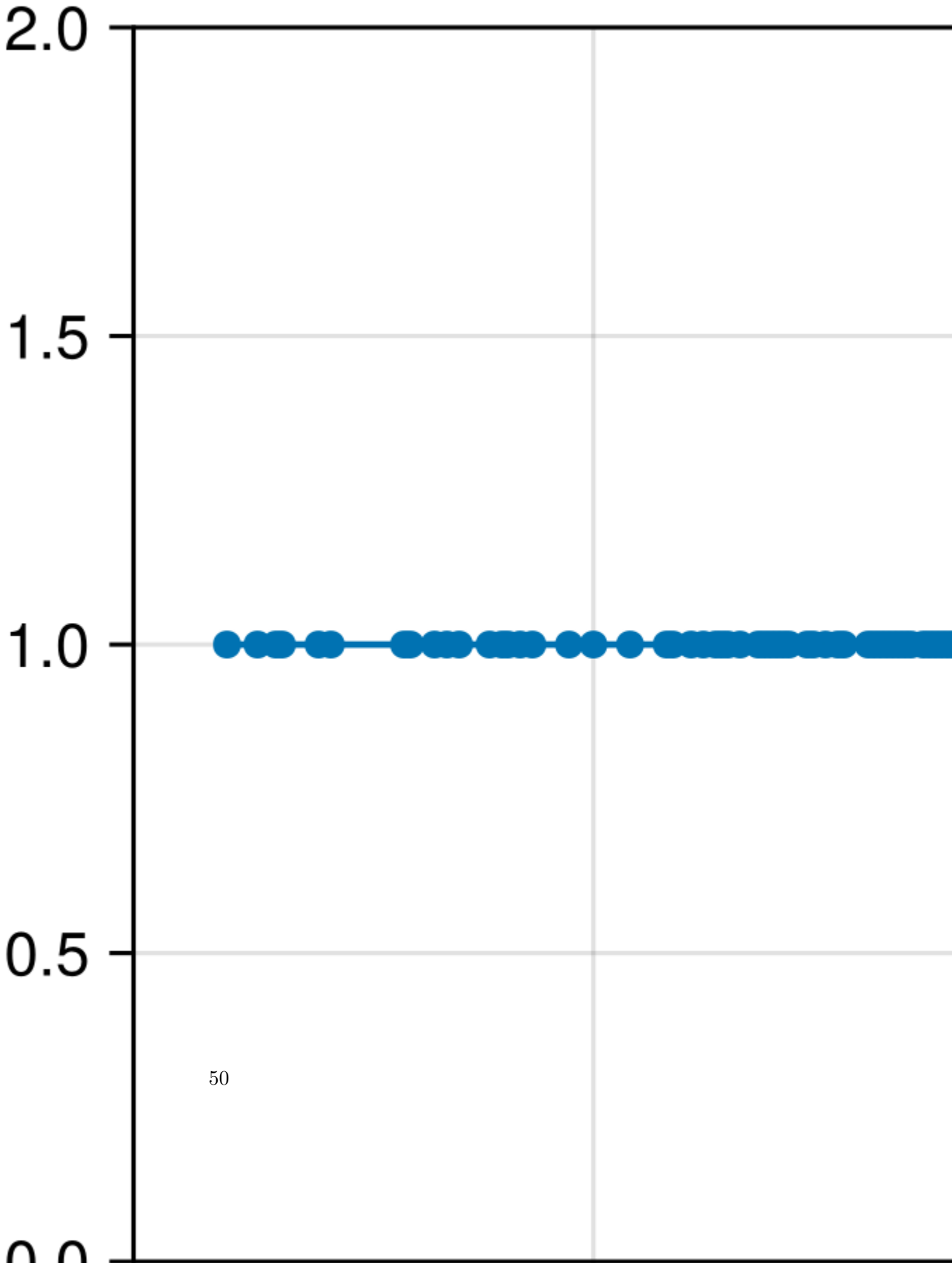
Boosting to create a BRT

TODO

Explaining an BRT's predictions

Partial Responses

```
variables(rf_sdm)
prt = partialresponse(rf_sdm, variables(rf_sdm)[6])
scatterlines(prt...)
```


2.0
1.5
1.0
0.5

Partial response in space

```
partial1 = partialresponse(rf_sdm, env_covariates, 1; threshold=false)
```

```
A 240 × 546 layer with 70065 Float64 cells
Projection: +proj=longlat +datum=WGS84 +no_defs
```

SHAP Values

Layer-wise

```
#shapley1 = explain(rf_sdm, env_covariates, 1; threshold=false)
```

All at once (this takes a while)

```
#@time S = explain(rf_sdm, env_covariates; threshold=false)
```

```
f = Figure(; size = (600, 300))
mostimp = mosaic(argmax, map(x -> abs.(x), S))
colmap = [colorant"#E69F00", colorant"#56B4E9", colorant"#009E73", colorant"#D55E00", colorant
ax = Axis(f[1, 1]; aspect = DataAspect())
heatmap!(ax, mostimp; colormap = colmap)
lines!(ax, switzerland; color = :black)
hidedecorations!(ax)
hidespines!(ax)
Legend(
    f[2, 1],
    [PolyElement(; color = colmap[i]) for i in 1:length(variables(rf_sdm))],
    ["BIO$(b)" for b in variables(rf_sdm)];
    orientation = :horizontal,
    nbanks = 1,
    framevisible = false,
    vertical = false,
)
f
```

Conformal Prediction

Conformal prediction is another method for interpreting machine learning models. Specifically, it is a form of *uncertainty quantification*.

Let's talk about uncertainty. A few sections ago, we generated a map of ucnertainty from our bagged model. This was done by measuring the interquantile range across each individual tree.

```
include("conformal_prediction.jl")
```

```
cellsize (generic function with 1 method)
```

```
rlevels = LinRange(0.01, 0.2, 100)
qs = [_estimate_q(rf_sdm, holdout(rf_sdm)...;  =u) for u in rlevels]
  = predict(rf_sdm; threshold=false)
eff = [mean(length.(credibleclasses.( , q))) for q in qs]
```

```
100-element Vector{Float64}:
 1.5382011605415862
 1.5382011605415862
 1.3786266924564796
 1.1731141199226305
 1.1731141199226305
 1.175531914893617
 1.1344294003868471
 1.4550290135396517
 1.2166344294003868
 1.1697292069632494

 0.9061895551257253
 0.8950676982591876
 0.8936170212765957
 0.8853965183752418
 0.9177949709864603
 0.9177949709864603
 0.8955512572533849
 0.9216634429400387
 0.8655705996131529
```

```
cs = cellsize(prd)
cmodel = deepcopy(rf_sdm)

# majority rules arg for consensus should give bool
distrib = predict(cmodel, env_covariates; consensus=majority, threshold=true)
for i in eachindex(qs)
    Cp, Ca = credibleclasses(prd, qs[i])
    undet = .!(Cp .| Ca)
    sure_presence = Cp .& (.!Ca)
    unsure = Ca .& Cp
    unsure_presence = unsure .& distrib
    unsure_absence = unsure .& (.!distrib)
end

# Cross-conformal with median range selected
q = median([_estimate_q(cmodel, fold...;  =0.05) for fold in kfold(cmodel; k=10)])
Cp, Ca = credibleclasses(prd, q)

# Partition
sure_presence = Cp .& (.!Ca)
sure_absence = Ca .& (.!Cp)
unsure = Ca .& Cp
unsure_in = unsure .& distrib
unsure_out = unsure .& (.!distrib)
```

```
A 240 × 546 layer with 70065 Bool cells
 Projection: +proj=longlat +datum=WGS84 +no_defs
```

```
f = Figure(; size=(1200, 600))
ax = Axis(f[1:2, 1]; aspect=DataAspect())
poly!(ax, switzerland, color=:grey95)
heatmap!(ax, nodata(sure_presence, false), colormap=[:forestgreen])
heatmap!(ax, nodata(unsure, false), colormap=[:grey70])
hidespines!(ax)
hidedecorations!(ax)
f
```

Counterfactuals

Appendix

What if I want to use a different model?

We can extract features and labels as a whole, but we can also use some more fine-grained functions to setup crossvalidation splits without inducing data leakage.