

## ✓ 2024 EY Data Science Challenge - Level 1

```

! pip install contextily
! pip install zarr
! pip install pystac
! pip install stackstac
! pip install pystac_client
! pip install planetary_computer

# Supress Warnings
import warnings
warnings.filterwarnings('ignore')

# Plotting
import matplotlib.pyplot as plt

# Data science
import pandas as pd
import numpy as np

# Machine Learning
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import f1_score, accuracy_score, ConfusionMatrixDisplay
from sklearn.model_selection import StratifiedKFold

# Geospatial
import contextily as cx
from shapely.geometry import Point, Polygon
import xarray as xr
import rasterio.features
import rasterio as rio
import fsspec
import zarr

# API
import requests
import json

# Import Planetary Computer
import stackstac
import pystac
import pystac_client
import planetary_computer

# Other
import os
from itertools import cycle

# # Folder to store extracted files
# storage_path = './files/output/'

# Path to data folder with provided material
data_path = '/content/drive/My Drive'

```

## ✓ Response Variable

Before we can build our model, we need to load in the frog occurrences data and generate our response variable. To do this, we first need to unzip the training data and store it on our machine. Then we can write a function that abstracts the loading process, with the option of providing a bounding box to only take those occurrences within a region of interest.

```
def get_frogs(file, year_range=None):
    """Returns the dataframe of all frog occurrences for the bounding box specified."""
    columns = [
        'gbifID', 'eventDate', 'country', 'continent', 'stateProvince',
        'decimalLatitude', 'decimalLongitude', 'species'
    ]
    country_names = {
        'AU': 'Australia', 'CR': 'Costa Rica', 'ZA': 'South Africa', 'MX': 'Mexico', 'HN': 'Honduras',
        'MZ': 'Mozambique', 'BW': 'Botswana', 'MW': 'Malawi', 'CO': 'Colombia', 'PA': 'Panama', 'NI': 'Nicaragua',
        'BZ': 'Belize', 'ZW': 'Zimbabwe', 'SZ': 'Eswatini', 'ZM': 'Zambia', 'GT': 'Guatemala', 'LS': 'Lesotho',
        'SV': 'El Salvador', 'AO': 'Angola', np.nan: 'unknown or invalid'
    }
    continent_names = {
        'AU': 'Australia', 'CR': 'Central America', 'ZA': 'Africa', 'MX': 'Central America', 'HN': 'Central America',
        'MZ': 'Africa', 'BW': 'Africa', 'MW': 'Africa', 'CO': 'Central America', 'PA': 'Central America',
        'NI': 'Central America', 'BZ': 'Central America', 'ZW': 'Africa', 'SZ': 'Africa', 'ZM': 'Africa',
        'GT': 'Central America', 'LS': 'Africa', 'SV': 'Central America', 'AO': 'Africa', np.nan: 'unknown or invalid'
    }
    frogs = (
        pd.read_csv(data_path + '/occurrence.txt', sep='\t', parse_dates=['eventDate'])
        .assign(
            country = lambda x: x.countryCode.map(country_names),
            continent = lambda x: x.countryCode.map(continent_names),
            species = lambda x: x.species.str.title()
        )
        [columns]
    )
    if year_range is not None:
        frogs = frogs[lambda x:
            (x.eventDate.dt.year >= year_range[0]) &
            (x.eventDate.dt.year <= year_range[1])
        ]
    # if bbox is not None:
    #     frogs = filter_bbox(frogs, bbox)
    return frogs
```

Sub-sampling

For this demonstration, we will constrain our search to frogs in the Australia found between the start of 2021 to the end of 2022. This gives a varied landscape of bushland, plains, rivers, and urban areas. This is done by providing `year_range` and `bbox` parameters to the `get_frogs` function we defined above.

```
from google.colab import drive
drive.mount('/content/drive')
# Load in data
all_frog_data = get_frogs(data_path + '/occurrence.txt', year_range=(2021, 2022))
all_frog_data
```

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/c

	gbifID	eventDate	country	continent	stateProvince	decimalLatitude
--	--------	-----------	---------	-----------	---------------	-----------------

4	3407962690	2021-10-29 13:57:00	Australia	Australia	Queensland	-26.71430;
10	3456121663	2021-12-22 20:51:56	Costa Rica	Central America	Puntarenas	9.41414;
16	3302372112	2021-05-05 10:56:00	Australia	Australia	South Australia	-35.20896;
19	3058997399	2021-03-01 17:50:00	Australia	Australia	Queensland	-26.43209;
22	3097032951	2021-03-30 15:53:47	Botswana	Africa	Central	-22.30180;
...	...	...	...	...	...	...

```
all_frog_data = all_frog_data[all_frog_data['country'] == 'Australia']
all_frog_data = all_frog_data[all_frog_data['stateProvince'] != 'Queensland ']
```

Addressing bias

Below we define some functions to assist in plotting the frog data. This will assist us in identifying two main areas of bias. We then use these functions to plot the frog species distributions of each country.

```
def plot_species(fig, ax, frog_data, region_name, cmap_params={'alpha':0.5}, scatter_params={'alpha':0.5}):
    # Bar chart
    bar_data = frog_data.species.value_counts()
    barchart = ax[1].bar(bar_data.index.str.replace(' ', '\n'), bar_data)

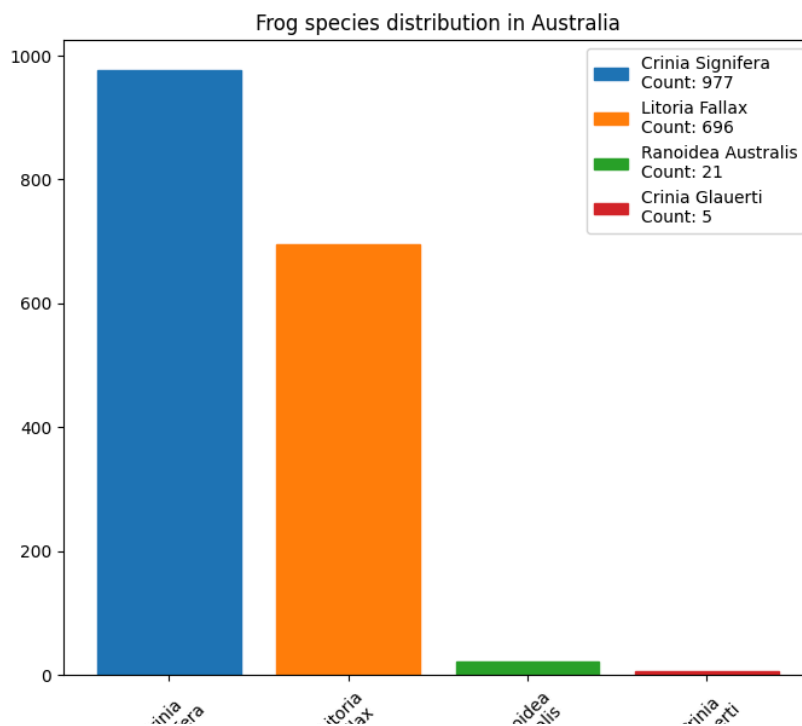
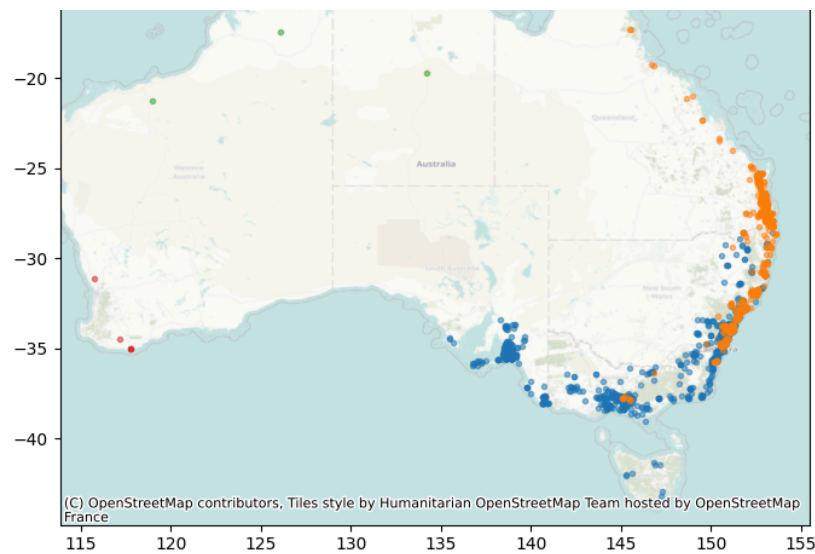
    # Colour cycle to ensure colors match in both plots
    prop_cycle = cycle(plt.rcParams['axes.prop_cycle'])
    for i, color in zip(range(len(bar_data)), prop_cycle):
        species_name = bar_data.index[i]
        if len(species_name) > 19:
            display_name = species_name.replace(' ', '\n')
        else:
            display_name = species_name
        barchart[i].set_color(color['color'])
        barchart[i].set_label(f"{display_name}\nCount: {bar_data[i]}")
        filt = frog_data.species == species_name
        # Scatter plot
        ax[0].scatter(
            frog_data[filt].decimalLongitude,
            frog_data[filt].decimalLatitude,
            marker='.',
            color=color['color'],
            **scatter_params
        )

    # Add other features
    ax[0].set_title(f"Frog occurrences for {region_name}")
    # ax[0].set_xticklabels([])
    # ax[0].set_yticklabels([])
    ax[1].set_title(f"Frog species distribution in {region_name}")
    cx.add_basemap(ax[0], crs={'init':'epsg:4326'}, **cmap_params) # Add basemap
    ax[1].set_xticklabels(bar_data.index.str.replace(' ', '\n'), rotation=45)
    ax[1].legend()

def plot_barchart(bar_data, ax, bar_params={}, fold_text=True):
    barchart = ax.bar(bar_data.index, bar_data, **bar_params)
    prop_cycle = cycle(plt.rcParams['axes.prop_cycle'])
    for i, color in zip(range(len(bar_data)), prop_cycle):
        var_name = bar_data.index[i]
        barchart[i].set_color(color['color'])
        barchart[i].set_label(f"{var_name}\nCount: {bar_data[i]}")
    ax.set_xticklabels(bar_data.index.str.replace(' ', '\n'), rotation=45)
    ax.legend()
```

Below, we can visualise the frog species distribution of the area.

```
fig, ax = plt.subplots(2, 1, figsize=(8, 15))
region_name = 'Australia'
plot_species(fig, ax, all_frog_data, region_name)
```



### ✓ Sampling bias

The plot above shows how frog occurrences are heavily biased around urban areas, where people are more likely to come across them. They also cluster tightly around towns, parks, bush trails etc.

One method of addressing the sampling bias inherent in the database is to use the occurrence points of other species as absence points for the target species. This is called pseudo-absence and is a common technique in species distribution modelling. This way, if a different species of frog has been sighted in a specific location, we can be more certain that the species we are trying to predict is not at that same location. Alternatively, if we just picked a random point where there are no frog occurrences, we cannot be certain that frogs are not in that location. It might just be that there are no walking tracks near that location, and therefore the frogs would not show up in our database.

For this notebook, we will use the other species as examples of *Litoria fallax*'s absence. We will alter our response variable to be `occurrenceStatus` which will take the value of 1 if the occurrence species is *Litoria fallax*, and 0 if the species is not *Litoria fallax* (i.e. *is crinia signifera*).

```
target_species = 'Litoria Fallax'

all_frog_data = (
    all_frog_data
    # Assign the occurrenceStatus to 1 for the target species and 0 for all other species.
    # as well as a key for joining (later)
    .assign(
        occurrenceStatus = lambda x: np.where(x.species == target_species, 1, 0)
    )
)
all_frog_data
```

	gbifID	eventDate	country	continent	stateProvince	decimalLatitude
4	3407962690	2021-10-29 13:57:00	Australia	Australia	Queensland	-26.714302
16	3302372112	2021-05-05 10:56:00	Australia	Australia	South Australia	-35.208964
19	3058997399	2021-03-01 17:50:00	Australia	Australia	Queensland	-26.432093
24	3314568431	2021-06-19 12:00:38	Australia	Australia	Tasmania	-43.232435
36	3384069347	2021-09-04 18:21:48	Australia	Australia	New South Wales	-32.959908
...	...	...	...	...	...	...

## ✓ Class Balancing

Another bias shown in the above visualisations is the class imbalance. To handle this, we will down-sample the absence points so that their numbers match that of the target species. Note that this is quite a naive approach as an isolated frog occurrence may be lost while clusters of occurrences are more likely to persist. This may not be ideal, as these isolated occurrences are often the most important data points available for frog conservationists. Developing a smarter sampling method may be a worthwhile pursuit when considering ways to improve your model.

The barcharts below show the response variable before and after the classes have been balanced.

```
target_species_frog_data = all_frog_data[all_frog_data.occurrenceStatus == 1]

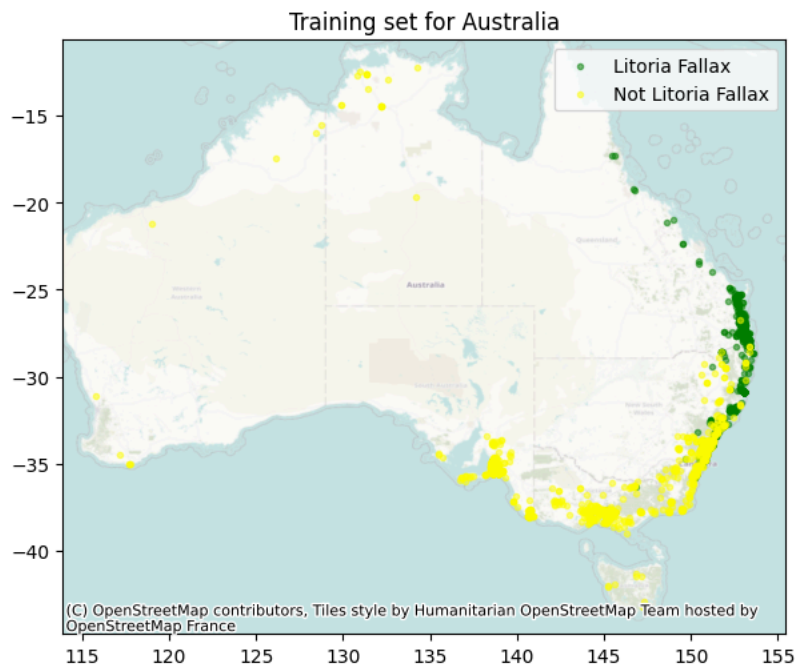
sampled_frog_data = all_frog_data[all_frog_data.occurrenceStatus == 0]
# Use concat instead of append
frog_data = pd.concat([sampled_frog_data, target_species_frog_data]).reset_index(drop=True).assign(key=lambda x: x.index)
```

After assigning absence points and balancing the classes, we finally have our training data visualised below. Again, it must be stressed that this is just one way of addressing the sampling bias and class imbalances inherent in the GBIF data and it does not address these biases completely.

```
fig, ax = plt.subplots(figsize = (7, 7))

filt = frog_data.occurrenceStatus == 1
ax.scatter(frog_data[filt].decimalLongitude, frog_data[filt].decimalLatitude,
           color = 'green', marker='.', alpha=0.5, label=target_species)
ax.scatter(frog_data[~filt].decimalLongitude, frog_data[~filt].decimalLatitude,
           color = 'yellow', marker='.', alpha=0.5, label=f"Not {target_species}")
ax.legend()
cx.add_basemap(ax, crs={'init':'epsg:4326'}, alpha=0.5)
ax.set_title(f"Training set for {region_name}")
```

```
Text(0.5, 1.0, 'Training set for Australia')
```



## ✓ Predictor Variables

### Accessing the TerraClimate Data

To get the TerraClimate data, we write a function called `get_terraclimate`. This function will fetch all data intersecting with the bounding box and will calculate various metrics over the time dimension for each coordinate. In this example, we will take five metrics from five assets, namely the mean maximum monthly air temp (`tmax_mean`), mean minimum monthly air temp (`tmin_mean`), mean accumulated precipitation (`ppt_mean`), mean soil moisture (`soil_mean`) and mean Actual Evapotranspiration (`aet_mean`) all calculated over a five year timeframe from the start of 2021 to the end of 2022.

To assist in visualisations, this function has an interpolation functionality which will allow the comparatively coarse temporal resolution of the terraclimate data to be mapped to a larger set of coordinates, creating an ( $n \times m$ ) image. We will choose (512 x 512).

```
longitude_min = all_frog_data['decimalLongitude'].min()
longitude_max = all_frog_data['decimalLongitude'].max()

latitude_min = all_frog_data['decimalLatitude'].min()
latitude_max = all_frog_data['decimalLatitude'].max()

min_lon, min_lat = (longitude_min, latitude_min) # Lower-left corner
max_lon, max_lat = (longitude_max, latitude_max) # Upper-right corner
bbox = (min_lon, min_lat, max_lon, max_lat)

print(f"经度最小值: {longitude_min}, 经度最大值: {longitude_max}")
print(f"纬度最小值: {latitude_min}, 纬度最大值: {latitude_max}")
```

```
经度最小值: 115.750686, 经度最大值: 153.580567
纬度最小值: -43.232435, 纬度最大值: -12.212202
```

```

def get_terraclimate(bbox, metrics, time_slice=None, assets=None, features=None, interp_dims=None, verbose=True):
    """Returns terraclimate metrics for a given area, allowing results to be interpolated onto a larger image.

    Attributes:
    bbox -- Tuple of (min_lon, min_lat, max_lon, max_lat) to define area
    metrics -- Nested dictionary in the form {<metric_name>:{'fn':<metric_function>,'params':<metric_kwargs_dict>}, ... }
    time_slice -- Tuple of datetime strings to select data between, e.g. ('2015-01-01','2019-12-31')
    assets -- list of terraclimate assets to take
    features -- list of asset metrics to take, specified by strings in the form '<asset_name>_<metric_name>'
    interp_dims -- Tuple of dimensions (n, m) to interpolate results to
    """

    min_lon, min_lat, max_lon, max_lat = bbox

    collection = pystac.read_file("https://planetarycomputer.microsoft.com/api/stac/v1/collections/terraclimate")
    asset = collection.assets["zarr-https"]
    store = fsspec.get_mapper(asset.href)
    data = xr.open_zarr(store)

    # Select datapoints that overlap region
    if time_slice is not None:
        data = data.sel(lon=slice(min_lon,max_lon),lat=slice(max_lat,min_lat),time=slice(time_slice[0],time_slice[1]))
    else:
        data = data.sel(lon=slice(min_lon,max_lon),lat=slice(max_lat,min_lat))
    if assets is not None:
        data = data[assets]
    print('Loading data') if verbose else None
    data = data.rename(lat='y', lon='x').to_array().compute()

    # Calculate metrics
    combined_values = []
    combined_bands = []
    for name, metric in metrics.items():
        print(f'Calculating {name}') if verbose else None
        sum_data = xr.apply_ufunc(
            metric['fn'], data, input_core_dims=[["time"]], kwargs=metric['params'], dask = 'allowed', vectorize = True
        ).rename(variable='band')
        xcoords = sum_data.x
        ycoords = sum_data.y
        dims = sum_data.dims
        combined_values.append(sum_data.values)
        for band in sum_data.band.values:
            combined_bands.append(band+'_'+name)

    # Combine metrics
    combined_values = np.concatenate(
        combined_values,
        axis=0
    )
    combined_data = xr.DataArray(
        data=combined_values,
        dims=dims,
        coords=dict(
            band=combined_bands,
            y=ycoords,
            x=xcoords
        )
    )

    # Take relevant bands:
    combined_data = combined_data.sel(band=features)

    if interp_dims is not None:
        print(f'Interpolating image') if verbose else None
        interp_coords = (np.linspace(bbox[0], bbox[2], interp_dims[0]), np.linspace(bbox[1], bbox[3], interp_dims[1]))
        combined_data = combined_data.interp(x=interp_coords[0], y=interp_coords[1], method='nearest', kwargs={"fill_value": "extrapola"})

    return combined_data

```

Below, we define the products to take from TerraClimate in `assets` and the metrics to calculate from them in `tc_metrics`. Each metric is applied to each asset, so to pick the desired asset/metric pairs we define a list of strings in the form '<asset>\_<metric>' in `features`.

```

# Metrics to measure over time dimension
tc_metrics = {
    'mean':{
        'fn':np.nanmean,
        'params':{}
    },
    'min':{
        'fn':np.nanmin,
        'params':{}
    },
    'max':{
        'fn':np.nanmax,
        'params':{}
    }
}

# Date range to take
time_slice = ('2018-01-01','2022-12-31')

# Measurements to take
assets=['tmax', 'tmin', 'ppt', 'soil', 'aet']

# Features to take, in form '<asset>_<metric>'
features=['tmax_mean', 'tmin_mean', 'ppt_mean', 'soil_mean', 'aet_mean']

# Interpolate values to a 512x512 image
interp_dims = (512, 512)

weather_data = get_terraclimate(bbox, tc_metrics, time_slice=time_slice, assets=assets, features=features, interp_dims=interp_dims)
display(weather_data.band.values)

Loading data
Calculating mean

```

## ✓ Visualising the TerraClimate Data

The spatial distribution of the four variables are displayed below.

```

nrow = 2
ncol = 3
fig, ax = plt.subplots(nrow, ncol, figsize=(13, 7), sharex=True, sharey=True)

bands = weather_data.band.values
filt = frog_data.occurrenceStatus == 1
cmaps = ["cool", "cool", "Blues", "BrBG"]*2

for i in range(len(bands)):
    xr.plot.imshow(weather_data[i], 'x', 'y', cmap=cmaps[i], ax=ax[i//ncol, i%ncol])
    ax[i//ncol, i%ncol].set_title(bands[i])
    ax[i//ncol, i%ncol].scatter(frog_data[filt].decimalLongitude, frog_data[filt].decimalLatitude,
                               color = 'yellow', marker='.', alpha=0.5, label=target_species if i==0 else '

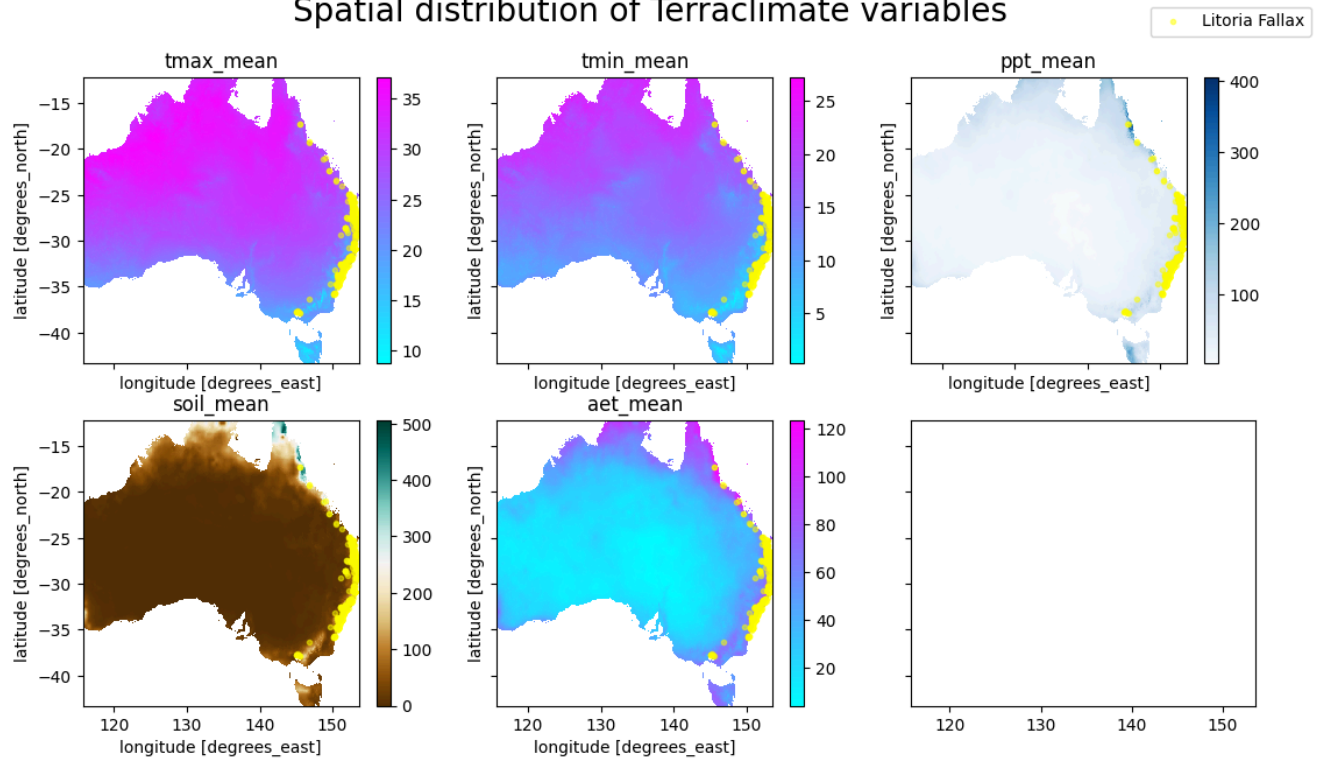
fig.suptitle("Spatial distribution of Terraclimate variables", fontsize=20)
fig.legend(loc=(0.85, 0.933))

```



&lt;matplotlib.legend.Legend at 0x7dd31ec58910&gt;

## Spatial distribution of Terraclimate variables



### Feature engineering

The frequency distribution of each variable is displayed below. There is some skewness present in a few variables, so you might want to address this when training your own model. Depending on the type of model you decide to train, some of the variables might require normalisation, standardisation, or transformation. For now, we will proceed with the variables as they come.

```
nrow = 2
ncol = 3
fig, ax = plt.subplots(nrow, ncol, figsize=(13, 7))

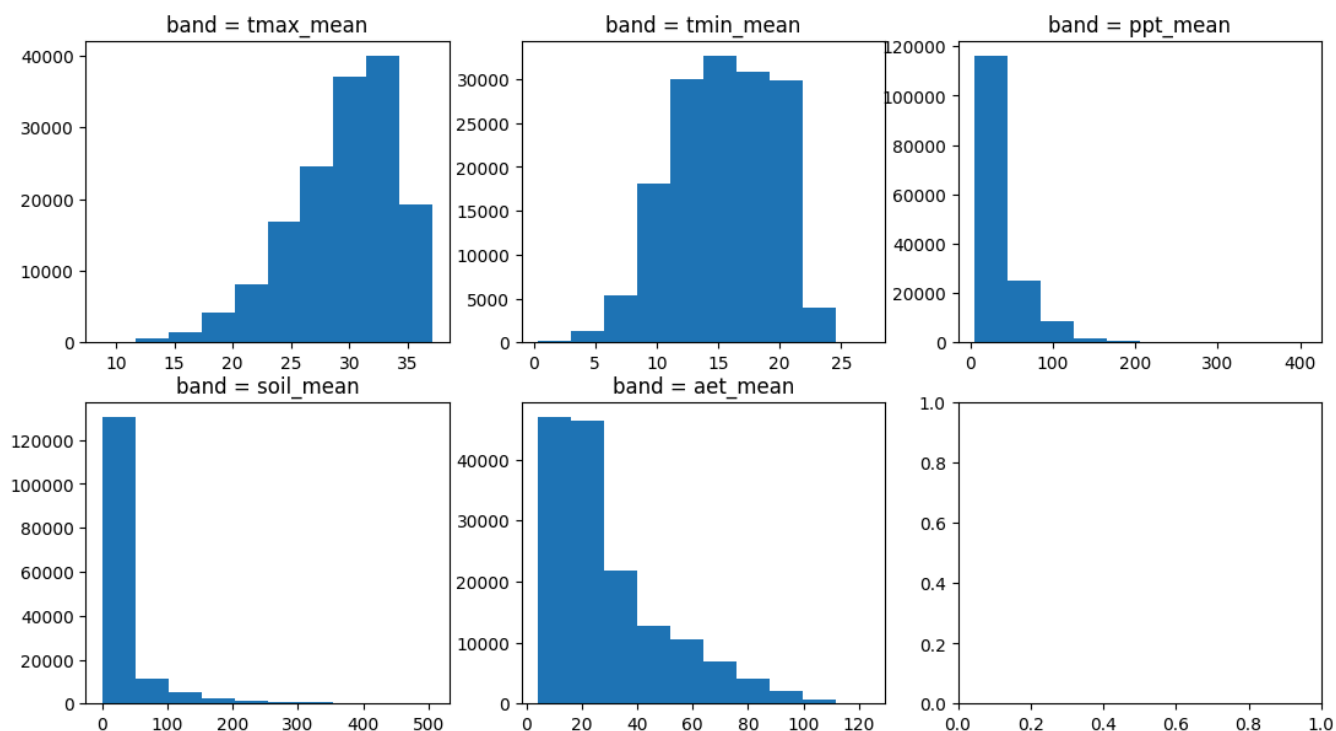
bands = weather_data.band.values

for i in range(len(bands)):
    xr.plot.hist(weather_data[i], ax=ax[i//ncol, i%ncol])

fig.suptitle("Frequency distribution of TerraClimate variables", fontsize=20)
```

Text(0.5, 0.98, 'Frequency distribution of TerraClimate variables')

## Frequency distribution of TerraClimate variables



### Joining Predictors to the Response Variable

Now that we have read in our predictor variables, we need to join them onto the response variable of frogs. To do this, we loop through the frog occurrence data and assign each frog occurrence the closest predictor pixel value from each of the predictor variables based on the geo-coordinates. The `sel` method of the xarray dataarray comes in handy here.

```
def join_frogs(frogs, data):
    """Collects the data for each frog location and joins it onto the frog data

    Arguments:
    frogs -- dataframe containing the response variable along with ["decimalLongitude", "decimalLatitude", "key"]
    data -- xarray dataarray of features, indexed with geocoordinates
    """
    return frogs.merge(
        (
            data
            .rename('data')
            .sel(
                x=xr.DataArray(frog_data.decimalLongitude, dims="key", coords={"key": frog_data.key}),
                y=xr.DataArray(frog_data.decimalLatitude, dims="key", coords={"key": frog_data.key}),
                method="nearest"
            )
            .to_dataframe()
            .assign(val = lambda x: x.iloc[:, -1])
            [['val']]
            .reset_index()
            .drop_duplicates()
            .pivot(index="key", columns="band", values="val")
            .reset_index()
        ),
        on = ['key'],
        how = 'inner'
    )

model_data = join_frogs(frog_data, weather_data)
model_data.head()
```

	gbifID	eventDate	country	continent	stateProvince	decimalLatitude	decimalLongitude	species	occurrenceStatus	key
0	3302372112	2021-05-05 10:56:00	Australia	Australia	South Australia	-35.208964	138.480985	Crinia Signifera	0	0
1	3314568431	2021-06-19 12:00:38	Australia	Australia	Tasmania	-43.232435	147.257886	Crinia Signifera	0	1

Model Building

Model Training

Scikit-learn models require separation of predictor variables and the response variable. We store the predictor variables in dataframe `X` and the response in the array `y`. We must make sure to drop the response variable from `X`, otherwise the model will have the answers! It also doesn't make sense to use latitude and longitude as predictor variables in such a confined area, so we drop those too.


```
from sklearn.ensemble import RandomForestClassifier

full_model = RandomForestClassifier(max_depth=None, min_samples_leaf=1, min_samples_split=2, n_estimators=100, random_state=42, class_weight='balanced')

X = model_data.drop(['gbifID', 'eventDate', 'decimalLatitude', 'decimalLongitude', 'species',
                    'country', 'continent', 'stateProvince', 'occurrenceStatus', 'key'], axis=1)
y = model_data.occurrenceStatus.astype(int)

X = X.apply(lambda x: x.fillna(x.mean()), axis=0)

full_model.fit(X, y)
```



RandomForestClassifier  
RandomForestClassifier(class\_weight='balanced', random\_state=42)

Model Prediction

Predict Training Set

Logistic regression is a machine learning model that estimates the probability of a binary response variable. In our case, the model will output the probability of a frog being present at a given location. To obtain the predictions for our training set, we simply use the `predict` method on our trained model. We will evaluate these predictions in the evaluation section of this notebook.

开始借助 AI 编写或生成代码。

```
predictions = full_model.predict(X)
```

Predict Entire Region

For a species distribution model to be effective, it must also be capable of performing predictions over the entire region, not just the points in our training set. To do this, we will define another function called `predict_frogs` that will take our interpolated predictor variable image in, along with our logistic regression model, and output the probabilities for each pixel in the region. We will visualise these predictions in a heatmap in the results section of this notebook.

This function will be used later to predict the test regions for the challenge.

```
def predict_frogs(predictor_image, model):
    """Returns a (1, n, m) xarray where each pixel value corresponds to the probability of a frog occurrence.

    Takes in the multi-band image outputted by the `create_predictor_image` function as well as the
    trained model and returns the predictions for each pixel value. Firstly, the $x$ and $y$ indexes
    in the predictor image are stacked into one multi-index $z=(x, y)$ to produce an $k\times n$
    array, which is the format required to feed into our logistic regression model. Then, the array
    is fed into the model, returning the model's predictions for the frog likelihood at each pixel.
    The predicted probabilities are then indexed by the same multi-index $z$ as before, which allows
    the array to be unstacked and returned as a one-band image, ready for plotting.

    Arguments:
    predictor_image -- (K, n, m) xarray, where K is the number of predictor variables.
    model -- sklearn model with K predictor variables.
    """
    # Stack up pixels so they are in the appropriate format for the model
    predictor_image = predictor_image.stack(z=("y", "x")).transpose()
    # Reorder variables to be in same order as model
    predictor_image = predictor_image.sel(band=model.feature_names_in_)
    # Location of null values so that we can skip them (prediction model will break if nulls are present)
    null_pixels = (np.sum(predictor_image.isnull(), axis=-1) > 0)
    # Empty probabilities array
    probabilities = np.zeros((len(null_pixels), 2))
    # Calculate probability for each non-null pixel point
    probabilities[~null_pixels] = model.predict_proba(
        predictor_image[~null_pixels]
    )
    # Set null pixels to a probability of null
    probabilities[null_pixels] = np.array([np.nan, np.nan])
    # Just take probability of frog (class=1)
    probabilities = probabilities[:,1]
    # Add the coordinates to the probabilities, saving them in an xarray
    resultant_image = xr.DataArray(
        data=probabilities,
        dims=['z'],
        coords=dict(
            z=predictor_image.z
        )
    )
    # Unstack the image
    resultant_image = resultant_image.unstack()
    return resultant_image

# Calculate probability for each pixel point
resultant_image = predict_frogs(weather_data, full_model)
```

## ✓ Model Evaluation

Now that we have trained our model and made some predictions, all that is left is to evaluate it. We will do this by first visualising the output of the model with a probability heatmap. Then, we will evaluate both its in-sample and out-of-sample performance using the training set we have generated.

## ✓ In-Sample Evaluation

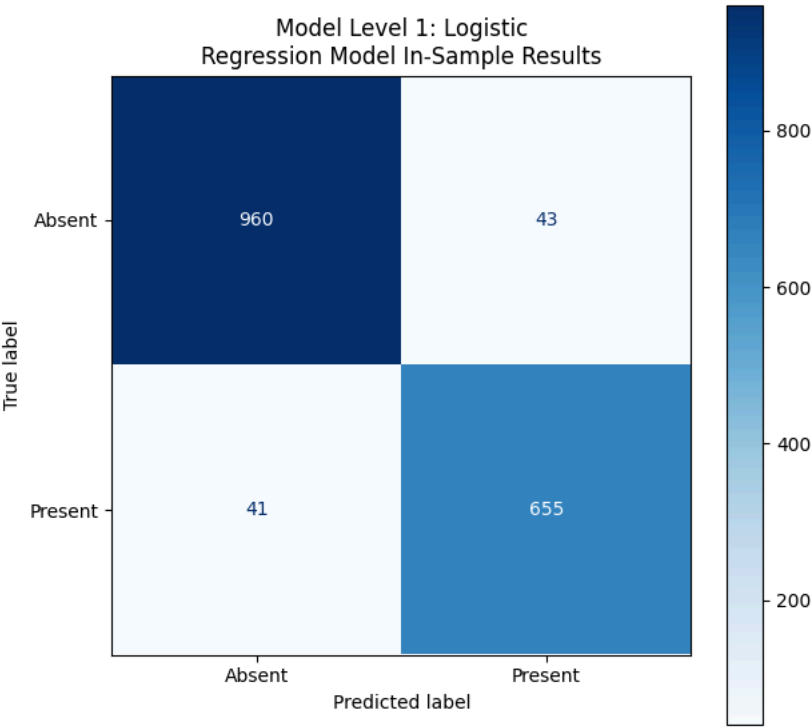
In the last section, we made our predictions for the training set and stored them in the `predictions` variable. We can now calculate some performance metrics to gauge the effectiveness of the model. It must be stressed that this is the in-sample performance - the performance on the training set. Hence, the values will tend to overestimate its performance. Additionally, the training set itself is biased and this notebook only took naive approaches to address this. The model evaluation metrics are only as good as the data used to evaluate it, so the metrics themselves will also be biased. Thus, these metrics are NOT truly indicative of this model's performance.

In this example, we will use `f1_score` and `accuracy_score` from Scikit-learn. Scikit-learn provides many other metrics that can be used for evaluation. You can even code your own if you think it will assist you in evaluating your model.

```
print(f"F1 Score: {np.mean(f1_score(y, predictions)).round(2)}")
print(f"Accuracy: {np.mean(accuracy_score(y, predictions)).round(2)}")

F1 Score: 0.94
Accuracy: 0.95

# Visualise the results in a confusion matrix
disp = ConfusionMatrixDisplay.from_estimator(full_model, X, y, display_labels=['Absent', 'Present'], cmap='Blues')
disp.figure_.set_size_inches((7, 7))
disp.ax_.set_title('Model Level 1: Logistic Regression Model In-Sample Results')
plt.show()
```



From above, we see that the model is able to achieve an ok F1 score and accuracy. From the confusion matrix, we can see that our model seems to confuse absent points with present points aka false positives, as shown in the top right corner. There may be many reasons for this, and a great way of understanding what might be causing the model's high false positive rate is to visualise its performance over the training region. We do this by plotting a probability heatmap in the section below.

Probability Heatmap

To create the probability heatmap, we write a function called `plot_heatmap`. This function will take in the model predictions from the entire region as stored in the `resultant_image` variable, and visualise these probabilities as a heatmap. In addition to the heatmap, we will also plot the actual map of the area in question, and the binary classification regions of the probability heatmap. The latter is simply a binary mask of the probability heatmap, 1 where the probability is greater than 0.5 and 0 elsewhere.

To help visualise the effectiveness of our model, we plot the target species occurrences over top of each image. This can give us an idea of where our model is doing well, and where it is doing poorly. Particularly, we are interested in the high false positive rate.

```

def plot_heatmap(resultant_image, frog_data, title, crs = {'init':'epsg:4326'}):
    """Plots a real map, probability heatmap, and model classification regions for the probability image from our model.

    Arguments:
    resultant_image -- (1, n, m) xarray of probabilities output from the model
    frog_data -- Dataframe of frog occurrences, indicated with a 1 in the occurrenceStatus column.
                  Must contain ["occurrenceStatus", "decimalLongitude", "decimalLatitude"]
    title -- string that will be displayed as the figure title
    crs -- coordinate reference system for plotting the real map. Defaults to EPSG:4326.
    """

    fig, ax = plt.subplots(1, 3, figsize = (20, 10), sharex=True, sharey=True)
    extent = [resultant_image.x.min(), resultant_image.x.max(), resultant_image.y.min(), resultant_image.y.max()]
    cmap = 'PiYG'

    # Plot real map
    ax[0].scatter(x=[extent[0], extent[1]], y=[extent[2], extent[3]], alpha=0)
    cx.add_basemap(ax[0], crs=crs)
    ax[0].set_title('Real map')

    # Plot heatmap from model
    heatmap = resultant_image.plot.imshow(
        x='x', y='y', ax=ax[1], cmap=cmap, vmin=0, vmax=1, interpolation='none', add_colorbar=False
    )
    ax[1].set_aspect('equal')
    ax[1].set_title('Model Probability Heatmap')

    # Plot binary classification from model
    regions = xr.where(resultant_image.isnull(), np.nan, resultant_image>0.5).plot.imshow(
        x='x', y='y', ax=ax[2], cmap=cmap, vmin=0, vmax=1, interpolation='none', add_colorbar=False
    )
    ax[2].set_aspect('equal')
    ax[2].set_title('Model Classification Regions')

    # Plot real frogs
    for i, axis in enumerate(ax):
        filt = frog_data.occurrenceStatus == 1
        axis.scatter(
            frog_data[filt].decimalLongitude, frog_data[filt].decimalLatitude,
            color = 'dodgerblue', marker='.', alpha=0.5, label='Target Species' if i==0 else ''
        )

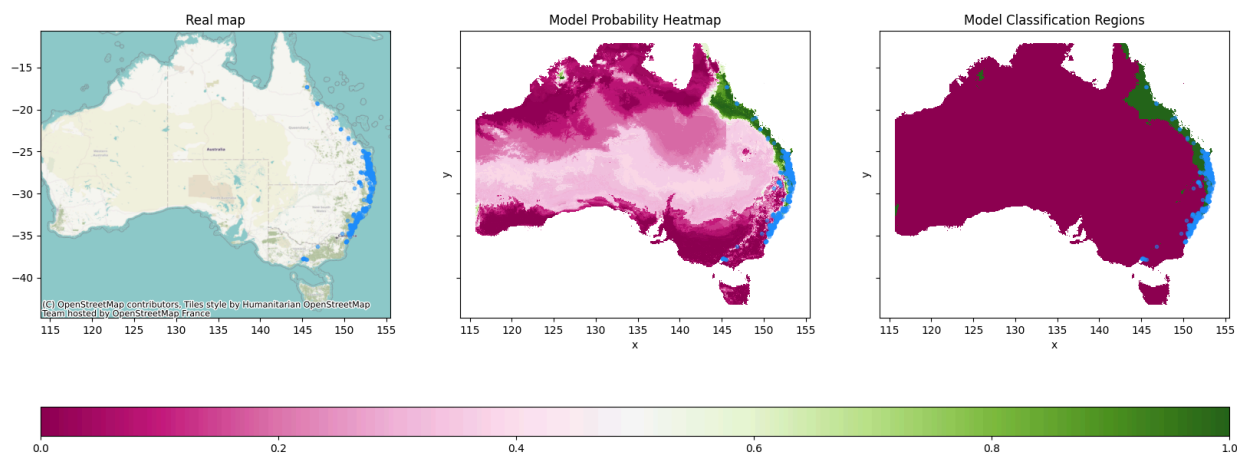
    fig.colorbar(heatmap, ax=ax, location = 'bottom', aspect=40)
    fig.legend(loc = (0.9, 0.9))
    fig.suptitle(title, x=0.5, y=0.9, fontsize=20)

plot_heatmap(resultant_image, frog_data, f"Logistic Regression Model Results - {region_name}")

```

## Logistic Regression Model Results - Australia

Target Species



From the plots above, we can see that the model does a pretty good job of mapping where *Litoria fallax* is. However, our performance metrics before suggested otherwise. The main limitation of the evaluation metrics comes from the pseudo absence species, *Crinia signifera*, sharing much of the same habitat as *Litoria fallax*. This paired with the relatively coarse spatial resolution of TerraClimate makes distinguishing close habitats difficult. This explains the high rate of false positives, as there are many frog absence points within the green classification region.

There are many ways you might go about addressing this issue. Perhaps choosing a species that does not closely share the same habitat as *Litoria fallax*. Alternatively, a greater variety of spatial sampling could also resolve this issue, i.e. picking extra regions where only *Litoria fallax* exists and some where only other species exist. Another option is to abandon the pseudo-absence approach entirely and develop a unique way of sampling for absence points. Ultimately, you might think it is worthwhile improving the training set to make your evaluation metrics a little more representative of the SDM performance.

## ✓ Out-of-Sample Evaluation

When evaluating a machine learning model, it is essential to correctly and fairly evaluate the model's ability to generalise. This is because models have a tendency to overfit the dataset they are trained on. To estimate the out-of-sample performance, we will use k-fold cross-validation. This technique involves splitting the training dataset into folds, in this case we will use 10. Each iteration, the model is trained on all but one of the folds, which is reserved for testing. This is repeated until all folds have been left out once. At the end of the process, we will have 10 metrics which can be averaged, giving a more reliable and valid measure of model performance.

Scikit-learn has built-in functions that can assist in k-fold cross validation. In particular, we will use `StratifiedKFold` to split our data into folds, ensuring there is always a balanced number of frogs and non-frogs in each fold.

Again, these metrics are derived from a biased sample, so be careful what you infer.

```

cv_model = RandomForestClassifier(max_depth=None, min_samples_leaf=1, min_samples_split=2, n_estimators=100, random_state=42, class_weight='balanced')

n_folds = 10

skf = StratifiedKFold(n_splits=n_folds, random_state=42, shuffle=True)
metrics = {'F1': f1_score, 'Accuracy': accuracy_score}
results = {'predicted': [], 'actual': []}
scores = {'F1': [], 'Accuracy': []}

for i, (train_index, test_index) in enumerate(skf.split(X, y)):
    # Split the dataset
    print(f"Fold {i+1} of {n_folds}")
    X_train, X_test = X.loc[train_index], X.loc[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Fit the model with the training set
    cv_model.fit(X_train, y_train)

    predictions = cv_model.predict(X_test)

    for metric, fn in metrics.items():
        scores[metric].append(fn(y_test, predictions))

    results['predicted'].extend(predictions)
    results['actual'].extend(list(y_test))

print(f"\nMetrics averaged over {n_folds} trials:")
for metric, result in scores.items():
    print(f"{metric}: {np.mean(result).round(2)}")

Fold 1 of 10
Fold 2 of 10
Fold 3 of 10
Fold 4 of 10
Fold 5 of 10
Fold 6 of 10
Fold 7 of 10
Fold 8 of 10
Fold 9 of 10
Fold 10 of 10

Metrics averaged over 10 trials:
F1: 0.88
Accuracy: 0.9

# Visualise the results in a confusion matrix
disp = ConfusionMatrixDisplay.from_predictions(results['actual'], results['predicted'], display_labels=['Absent', 'Present'], cmap='Blues')
disp.figure_.set_size_inches((7, 7))
disp.ax_.set_title('Model Level 1: Logistic Regression Model\n10-fold Cross Validation Results')
plt.show()

```

