Fabian_Gobet_SM_assignment6

June 25, 2024

0.1 BiSSe - Binary State Speciation and Extinction

The model described is a Birth-Death model with two interacting species and the possibility of transitions between them. This model captures the dynamics of two species populations over time, incorporating birth, death, and transition rates.

0.1.1 Parameters

- λ_1, λ_2: Birth rates of species 1 and species 2, respectively.
- μ_1, μ_2: Death rates of species 1 and species 2, respectively.
- p_{12}, p_{21}: Transition rates from species 1 to species 2 and from species 2 to species 1, respectively.
- ini_1, ini_2: Initial populations of species 1 and species 2, respectively.
- T: Maximum time for the simulation.

0.1.2 Model Dynamics

1. **Initialization**: The initial populations of species 1 and species 2 are set based on the given parameters `ini1` and `ini2`. The current time is initialized to zero.

2. **Event Simulation**: The process continues in a loop until the current time exceeds the maximum time `T` or both species' populations become zero.

   - **Total Rate Calculation**: At each step, the total rate of events is calculated as the sum of all possible events' rates:

     $$\text{total\_rate} = n_1(\lambda_1 + \mu_1 + p_{12}) + n_2(\lambda_2 + \mu_2 + p_{21})$$

     where n_1 and n_2 are the current populations of species 1 and species 2, respectively.

   - **Event Time Sampling**: The time until the next event is sampled from an exponential distribution with the rate parameter `total_rate`.

   - **Event Type Sampling**: The type of event is determined by sampling from a discrete distribution with probabilities proportional to the rates of each event:

     $$\text{event\_probs} = \left[\frac{n_1\lambda_1}{\text{total\_rate}}, \frac{n_2\lambda_2}{\text{total\_rate}}, \frac{n_1\mu_1}{\text{total\_rate}}, \frac{n_2\mu_2}{\text{total\_rate}}, \frac{n_1 p_{12}}{\text{total\_rate}}, \frac{n_2 p_{21}}{\text{total\_rate}}\right]$$

   - **Event Execution**: Based on the sampled event type, the populations are updated accordingly:

- **Birth of species 1**: $(n_1 \leftarrow n_1 + 1)$
- **Birth of species 2**: $(n_2 \leftarrow n_2 + 1)$
- **Death of species 1**: $(n_1 \leftarrow n_1 - 1)$
- **Death of species 2**: $(n_2 \leftarrow n_2 - 1)$
- **Transition from species 1 to species 2**: $(n_1 \leftarrow n_1 - 1), (n_2 \leftarrow n_2 + 1)$
- **Transition from species 2 to species 1**: $(n_2 \leftarrow n_2 - 1), (n_1 \leftarrow n_1 + 1)$

- **Event Recording**: Each event, along with the current time and updated populations, is recorded.

3. **Termination**: The process stops when the current time exceeds the maximum time `T` or both species' populations reach zero.

0.1.3 Output

The function returns a list of events, each represented as a tuple `(time, n1, n2)`, where `time` is the time of the event, and `n1` and `n2` are the populations of species 1 and species 2 after the event.

$$\text{total\_rate} = n_1(\lambda_1 + \mu_1 + p_{12}) + n_2(\lambda_2 + \mu_2 + p_{21})$$

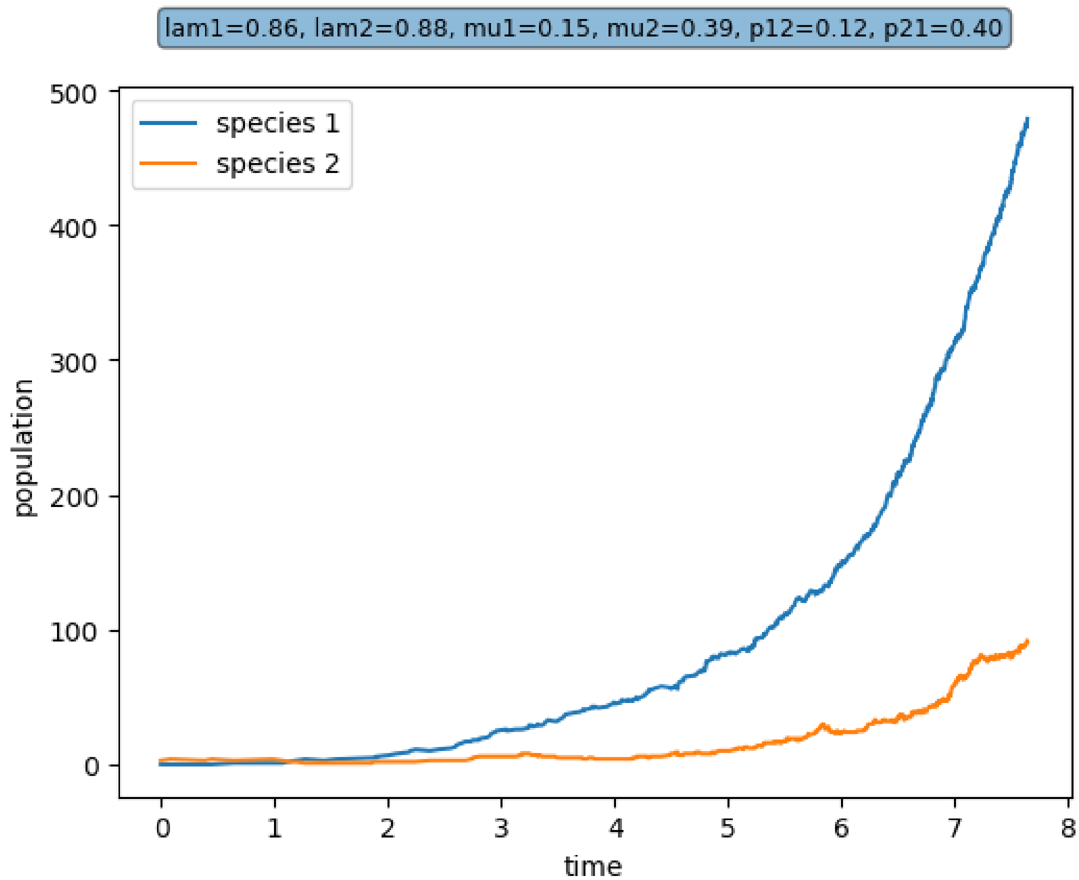
[ ]:
```
import numpy as np

def bisse(lam1, lam2, mu1, mu2, p12, p21, ini1, ini2, T, limit_event_size =␣
↪1000):

    n1 = ini1.copy()
    n2 = ini2.copy()
    current_time = 0
    events = []
    events_list = np.array([1,2,3,4,5,6])
    final_T = T

    while current_time < T:

        total_population = n1 + n2
        if total_population == 0:
            break
        if len(events) > limit_event_size:
            final_T = current_time
            break


        total_rate  = n1*(lam1+mu1+p12) + n2*(lam2+mu2+p21)
        sampled_time = np.random.exponential(1/total_rate)
        current_time += sampled_time

        if current_time > T:
            break
```

```
        event_probs = np.array([n1*lam1, n2*lam2, n1*mu1, n2*mu2, n1*p12,␣
↪n2*p21])/total_rate
        event = np.random.choice(events_list, p=event_probs)

        match event:
            case 1: # specie 1 gives birth
                n1 += 1
            case 2: # specie 2 gives birth
                n2 += 1
            case 3: # specie 1 dies
                n1 -= 1
            case 4: # specie 2 dies
                n2 -= 1
            case 5: # specie 1 transitions to specie 2
                n1 -= 1
                n2 += 1
            case 6: # specie 2 transitions to specie 1
                n2 -= 1
                n1 += 1
            case _:
                raise ValueError("Invalid event")

        events.append((current_time, n1, n2))

    return final_T, events
```

0.1.4 We randomize the paramaters and make an experiment by computing BiSSe and then ploting the evolution of the species over time

[ ]:
```
# parameters
lam1, lam2 = np.random.uniform(0, 1, size=2)
mu1, mu2 = np.random.uniform(0, 0.8, size=2)
p12, p21 = np.random.uniform(0, 0.5, size=2)
max_time = 10
max_num_initial_population = 5
ini1, ini2 = np.random.randint(0, max_num_initial_population, size=2)

final_T, events = bisse(lam1, lam2, mu1, mu2, p12, p21, ini1, ini2, max_time)
nodes = [(0, ini1, ini2)] + events

# plot each of the species in the same figure
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.plot([node[0] for node in nodes], [node[1] for node in nodes],␣
↪label='species 1')
```

```
ax.plot([node[0] for node in nodes], [node[2] for node in nodes],␣
 ↪label='species 2')
ax.set_xlabel('time')
ax.set_ylabel('population')
rates_info = f'lam1={lam1:.2f}, lam2={lam2:.2f}, mu1={mu1:.2f}, mu2={mu2:.2f},␣
 ↪p12={p12:.2f}, p21={p21:.2f}'
ax.text(0.05, 1.1, rates_info, transform=ax.transAxes, fontsize=9,␣
 ↪verticalalignment='top', bbox=dict(boxstyle="round", alpha=0.5))
ax.legend()
plt.show()
```



0.2 Function Description

The `generate_data` function simulates the dynamics of two interacting species over multiple iterations and collects the resulting data. This function leverages the Birth-Death model with transition rates between the species to generate the data points.

0.2.1 Parameters

- **num_data_points**: The number of data points to generate.
- **max_lamb_rate**: The maximum value for the birth rates λ_1 and λ_2.
- **max_mniu_rate**: The maximum value for the death rates μ_1 and μ_2.
- **max_num_initial_population**: The maximum initial population size for both species.
- **max_time**: The maximum simulation time for each iteration.

0.2.2 Function Dynamics

1. **Initialization**: Two empty lists, `X` and `Y`, are created to store the input parameters and the resulting populations, respectively.
2. **Loop through Data Points**: For each data point:
   - Randomly sample birth rates λ_1 and λ_2 from a uniform distribution between 0 and `max_lamb_rate`.
   - Randomly sample death rates μ_1 and μ_2 from a uniform distribution between 0 and `max_mniu_rate`.
   - Randomly sample transition rates p_{12} and p_{21} from a uniform distribution between 0 and 1.
   - Randomly sample a simulation time `time` from a uniform distribution between 0 and `max_time`.
   - Randomly sample initial populations `ini1` and `ini2` from an integer uniform distribution between 0 and `max_num_initial_population`.
3. **Simulation**: For each set of sampled parameters, the `bisse` function is called to simulate the population dynamics over the sampled time period.
4. **Event Recording**: The populations of species 1 and species 2 at the end of the simulation are recorded. If no events occurred during the simulation, the initial populations are used.
5. **Data Collection**: The sampled parameters and the resulting populations are appended to the lists `X` and `Y`.
6. **Return Values**: The function returns two NumPy arrays, `X` and `Y`, where `X` contains the input parameters for each data point and `Y` contains the resulting populations of species 1 and species 2.

[ ]:
```
from tqdm.notebook import tqdm

def generate_data(num_data_points, max_lamb_rate, max_mniu_rate,␣
 ↪max_num_initial_population, max_time):
    X = []
    Y = []

    for _ in tqdm(range(num_data_points)):
        lam1, lam2 = np.random.uniform(0, max_lamb_rate, size=2)
        mu1, mu2 = np.random.uniform(0, max_mniu_rate, size=2)
        p12, p21 = np.random.uniform(0, 1, size=2)
        time = np.random.uniform(0, max_time)
```

```
        ini1, ini2 = np.random.randint(0, max_num_initial_population, size=2)

        final_T, events = bisse(lam1, lam2, mu1, mu2, p12, p21, ini1, ini2,␣
 ↪time)

        _, num_specie1, num_specie2 = (0, ini1, ini2) if len(events) == 0 else␣
 ↪events[-1]

        X.append([lam1, lam2, mu1, mu2, p12, p21, ini1, ini2, final_T])
        Y.append([num_specie1, num_specie2])

    return np.array(X), np.array(Y)
```

0.3 Data Generation and Splitting

0.3.1 Data Generation

The data generation process involves simulating the dynamics of two interacting species over a large number of iterations using the `generate_data` function. The parameters for this process are as follows:

- **num_data_points**: (64×1250)
- **max_lamb_rate**: 1
- **max_mniu_rate**: 1
- **max_num_initial_population**: 5
- **max_time**: 10

The function `generate_data` is called with these parameters to produce the input data `X` and the corresponding output data `Y`.

Note: - We consider a small time frame of max time 10 because if sampled lambdas are substantially greater than the sampled mnius, then the growth of the population is exponential and this will become computationally unfeasible.

[ ]:
```
import pandas as pd
from sklearn.model_selection import train_test_split

num_data_points = 64*1250
max_lamb_rate = 1
max_mniu_rate = 1
max_num_initial_population = 5
max_time = 10

X, Y = generate_data(num_data_points, max_lamb_rate, max_mniu_rate,␣
 ↪max_num_initial_population, max_time)
# concatenate X and Y so values of Y are now last columns of X
data = np.concatenate([X, Y], axis=1)
#df_X = pd.DataFrame(X, columns=['lam1', 'lam2', 'mu1', 'mu2', 'p12', 'p21',␣
 ↪'ini1', 'ini2', 'final_T'])
```

```
#df_Y = pd.DataFrame(Y, columns=['num_specie1', 'num_specie2'])
#df = pd.concat([df_X, df_Y], axis=1)


train_set, test_set = train_test_split(data, test_size=0.1)
train_set, val_set = train_test_split(train_set, test_size=0.2)
```

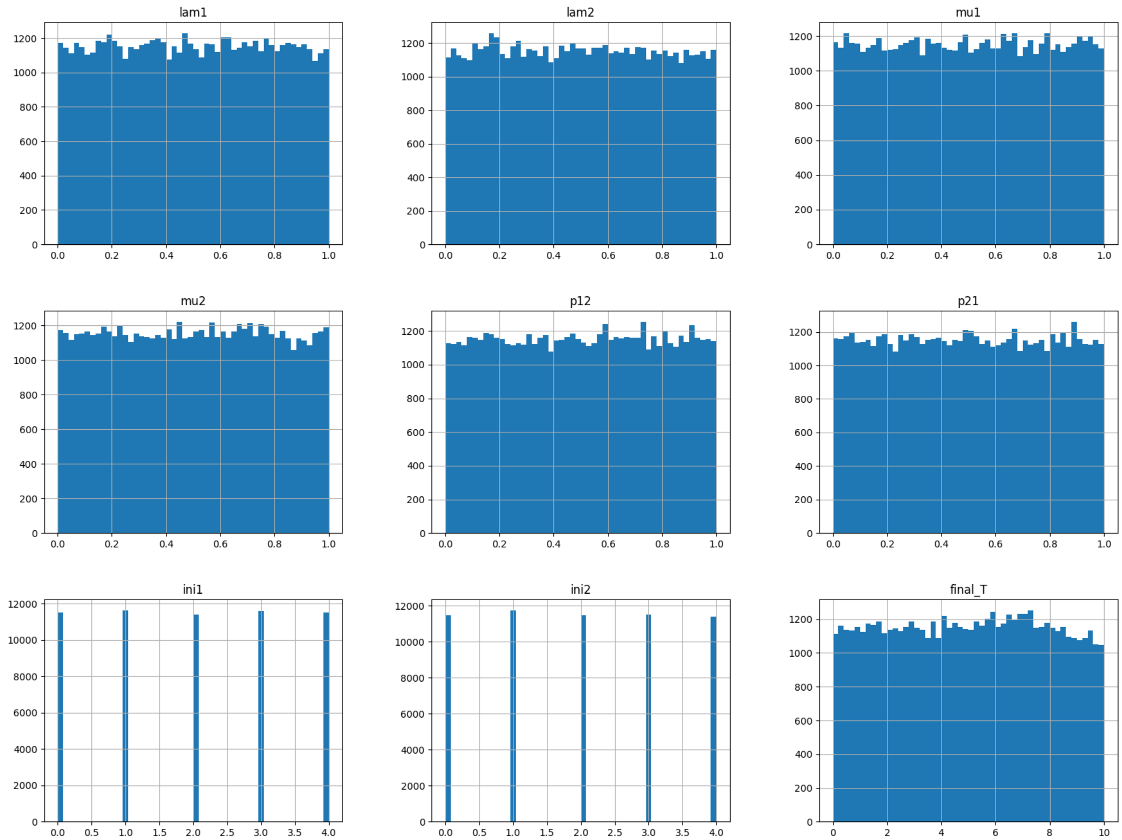
```
  0%|          | 0/80000 [00:00<?, ?it/s]
```

[ ]:
```
df = pd.DataFrame(train_set[:,:-2], columns=['lam1', 'lam2', 'mu1', 'mu2',␣
 ↪'p12', 'p21', 'ini1', 'ini2', 'final_T'])
df.corr()
```

[ ]:
```
            lam1      lam2       mu1       mu2       p12       p21      ini1  \
lam1     1.000000 -0.003087  0.000468 -0.000927  0.001793 -0.008315 -0.001816
lam2    -0.003087  1.000000  0.000570  0.003218 -0.008575 -0.011794  0.005866
mu1      0.000468  0.000570  1.000000  0.005145 -0.001064  0.000233  0.004187
mu2     -0.000927  0.003218  0.005145  1.000000 -0.001248  0.006256  0.003015
p12      0.001793 -0.008575 -0.001064 -0.001248  1.000000  0.005505 -0.003973
p21     -0.008315 -0.011794  0.000233  0.006256  0.005505  1.000000 -0.001206
ini1    -0.001816  0.005866  0.004187  0.003015 -0.003973 -0.001206  1.000000
ini2     0.008502  0.002408 -0.001825 -0.004654 -0.000091 -0.000427 -0.003204
final_T -0.001754 -0.004721  0.006384  0.006904 -0.001251 -0.006715 -0.004029

            ini2   final_T
lam1     0.008502 -0.001754
lam2     0.002408 -0.004721
mu1     -0.001825  0.006384
mu2     -0.004654  0.006904
p12     -0.000091 -0.001251
p21     -0.000427 -0.006715
ini1    -0.003204 -0.004029
ini2     1.000000 -0.000614
final_T -0.000614  1.000000
```

0.3.2 Before we apply any transformation, we should see if distribution of each attribute is heavy tail.

[ ]:
```
import matplotlib.pyplot as plt
df.hist(bins=50, figsize=(20, 15))
plt.show()
```



0.3.3 From above we can see the distribution is uniform

0.3.4 Now we can start building the pipeline

[ ]:
```python
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import MinMaxScaler
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.utils.validation import check_array, check_is_fitted

class CustomTransformation(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.columns_ = ['lam1', 'lam2', 'mu1', 'mu2', 'p12', 'p21', 'ini1',␣
↪'ini2', 'final_T']

    def fit(self, X, y=None):
        X = check_array(X)
        assert X.shape[1] == len(self.columns_)
        self.n_features_in_ = X.shape[1]
        return self
```

```
    def transform(self, X):
        check_is_fitted(self)
        return pd.DataFrame(X, columns=self.columns_)  # Corrected line

    def get_feature_names_out(self, input_features=None):
        return self.columns_

pipeline = Pipeline([
    ('custom_transformation', CustomTransformation()),
    ('imputer', SimpleImputer(strategy='mean')),
    ('min_max_scaler', MinMaxScaler(feature_range=(-1, 1)))
])

train_X, val_X, test_X = train_set[:,:-2], val_set[:,:-2], test_set[:,:-2]
train_Y_prepared, val_Y_prepared, test_Y_prepared = train_set[:,-2:], val_set[:
 ↪,-2:], test_set[:,-2:]

train_X_prepared = pipeline.fit_transform(train_X)
val_X_prepared = pipeline.transform(val_X)
test_X_prepared = pipeline.transform(test_X)
```

0.4 Neural Network Model Training and Visualization

0.4.1 Neural Network Model Architecture

The code defines a neural network model using TensorFlow's Keras API. The model architecture consists of: - **Input Layer**: Defined by the shape of `X_train[0]`, which corresponds to the shape of the input data. - **Dense Layers**: Four hidden layers with 16, 26, 18, and 8 neurons respectively, each using ReLU (Rectified Linear Unit) activation function. - **Output Layer**: An output layer with neurons equal to the number of outputs (`Y_train[0].shape[0]`), which predicts the populations of species 1 and species 2.

0.4.2 Callback

- **Early Stopping**: A callback (`ea_callback`) is used to monitor the validation loss (`val_loss`). Training will stop early if the validation loss does not improve for 5 consecutive epochs (`patience=5`). The model will restore the weights that give the best validation loss (`restore_best_weights=True`).

0.4.3 Model Compilation

The model is compiled using the Adam optimizer (`optimizer='adam'`) and mean squared error (`loss='mse'`) as the loss function. The accuracy metric is used for evaluation (`metrics=['accuracy']`).

0.4.4 Model Training

The `model.fit` method is called to train the model: - **X_train, Y_train**: Training data and labels. - **epochs**: Number of epochs set to 25. - **batch_size**: Batch size set to 32. - **valida-**

tion_data: Validation data and labels provided as (X_val, Y_val). - **callbacks**: Early stopping callback (ea_callback) is passed to monitor validation loss during training.

0.4.5 Training History Visualization

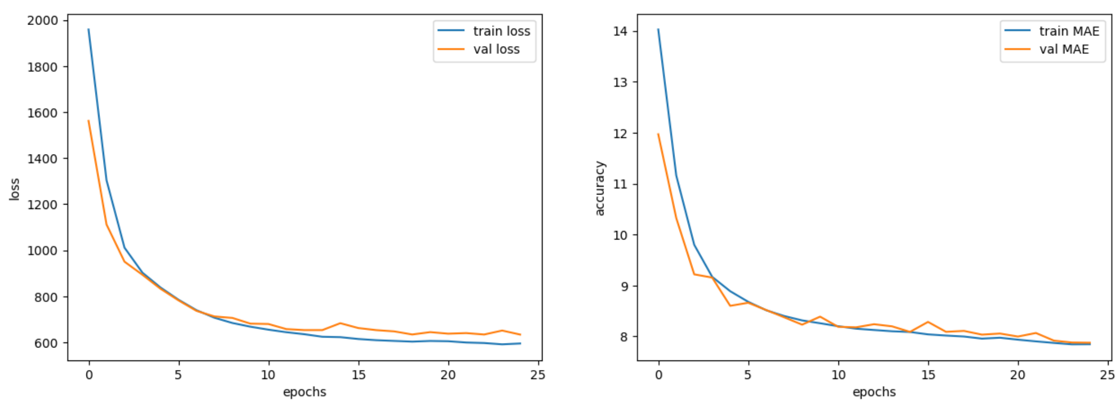
After training, the accuracy and validation accuracy over epochs are plotted using Matplotlib to visualize the model's performance.

[ ]:
```
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.InputLayer(train_X_prepared[0].shape),
    tf.keras.layers.Dense(16, activation='relu'),
    tf.keras.layers.Dense(22, activation='relu'),
    #tf.keras.layers.Dense(26, activation='relu'),
    #tf.keras.layers.Dense(18, activation='relu'),
    tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dense(train_Y_prepared[0].shape[0])
])

ea_callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5,␣
 ↪restore_best_weights=True)

model.compile(optimizer='adam', loss='mse', metrics=['mae'])
history = model.fit(train_X_prepared, train_Y_prepared, epochs=25,␣
 ↪batch_size=32, validation_data=(val_X_prepared, val_Y_prepared),␣
 ↪callbacks=[ea_callback])


fig, ax = plt.subplots(1, 2, figsize=(15, 5))

ax[0].plot(history.history['loss'], label='train loss')
ax[0].plot(history.history['val_loss'], label='val loss')
ax[0].set_xlabel('epochs')
ax[0].set_ylabel('loss')
ax[0].legend()

ax[1].plot(history.history['mae'], label='train MAE')
ax[1].plot(history.history['val_mae'], label='val MAE')
ax[1].set_xlabel('epochs')
ax[1].set_ylabel('accuracy')
ax[1].legend()

plt.show()
```

```
Epoch 1/25
1800/1800            1s 365us/step -
loss: 2261.5833 - mae: 14.3152 - val_loss: 1561.9283 - val_mae: 11.9693
```

```
Epoch 2/25
1800/1800            1s 328us/step -
loss: 1525.8071 - mae: 12.0507 - val_loss: 1112.0200 - val_mae: 10.3245
Epoch 3/25
1800/1800            1s 325us/step -
loss: 1104.9508 - mae: 10.2761 - val_loss: 950.6395 - val_mae: 9.2185
Epoch 4/25
1800/1800            1s 330us/step -
loss: 923.9717 - mae: 9.3391 - val_loss: 893.6246 - val_mae: 9.1550
Epoch 5/25
1800/1800            1s 335us/step -
loss: 885.6999 - mae: 9.1650 - val_loss: 833.1846 - val_mae: 8.6005
Epoch 6/25
1800/1800            1s 322us/step -
loss: 784.4977 - mae: 8.7070 - val_loss: 782.8658 - val_mae: 8.6613
Epoch 7/25
1800/1800            1s 324us/step -
loss: 714.7819 - mae: 8.3948 - val_loss: 737.2977 - val_mae: 8.5182
Epoch 8/25
1800/1800            1s 321us/step -
loss: 709.4855 - mae: 8.4658 - val_loss: 712.8978 - val_mae: 8.3759
Epoch 9/25
1800/1800            1s 324us/step -
loss: 689.1853 - mae: 8.3079 - val_loss: 706.5100 - val_mae: 8.2310
Epoch 10/25
1800/1800            1s 323us/step -
loss: 668.0004 - mae: 8.2740 - val_loss: 681.7399 - val_mae: 8.3883
Epoch 11/25
1800/1800            1s 342us/step -
loss: 669.3351 - mae: 8.1972 - val_loss: 680.1402 - val_mae: 8.1894
Epoch 12/25
1800/1800            1s 343us/step -
loss: 644.2637 - mae: 8.0893 - val_loss: 658.0243 - val_mae: 8.1791
Epoch 13/25
1800/1800            1s 339us/step -
loss: 606.0287 - mae: 8.0058 - val_loss: 653.7067 - val_mae: 8.2394
Epoch 14/25
1800/1800            1s 326us/step -
loss: 645.1565 - mae: 8.2028 - val_loss: 653.6976 - val_mae: 8.1980
Epoch 15/25
1800/1800            1s 326us/step -
loss: 644.7923 - mae: 8.1552 - val_loss: 683.6093 - val_mae: 8.0892
Epoch 16/25
1800/1800            1s 330us/step -
loss: 644.0101 - mae: 8.1606 - val_loss: 662.6105 - val_mae: 8.2848
Epoch 17/25
1800/1800            1s 339us/step -
loss: 614.5897 - mae: 8.0538 - val_loss: 653.5740 - val_mae: 8.0905
```

```
Epoch 18/25
1800/1800             1s 326us/step -
loss: 650.6334 - mae: 8.1484 - val_loss: 648.0974 - val_mae: 8.1084
Epoch 19/25
1800/1800             1s 327us/step -
loss: 588.6266 - mae: 7.9054 - val_loss: 634.6246 - val_mae: 8.0349
Epoch 20/25
1800/1800             1s 323us/step -
loss: 651.9552 - mae: 8.2438 - val_loss: 644.8616 - val_mae: 8.0567
Epoch 21/25
1800/1800             1s 321us/step -
loss: 592.2283 - mae: 7.9120 - val_loss: 638.1567 - val_mae: 7.9978
Epoch 22/25
1800/1800             1s 337us/step -
loss: 590.5286 - mae: 7.9280 - val_loss: 640.6867 - val_mae: 8.0675
Epoch 23/25
1800/1800             1s 331us/step -
loss: 603.6562 - mae: 7.8389 - val_loss: 634.5442 - val_mae: 7.9178
Epoch 24/25
1800/1800             1s 320us/step -
loss: 605.6814 - mae: 7.9561 - val_loss: 651.7853 - val_mae: 7.8808
Epoch 25/25
1800/1800             1s 322us/step -
loss: 595.5242 - mae: 7.9243 - val_loss: 634.2979 - val_mae: 7.8763
```



0.5 Model Evaluation on Test Data

0.5.1 Model Evaluation

To evaluate the trained neural network model on the test data (`X_test` and `Y_test`), the `model.evaluate` method is used. This method computes the loss and metrics (accuracy in this case) on the test set.

[ ]:
```
cost, acc = model.evaluate(test_X_prepared, test_Y_prepared)
print(f'Test MAE: {acc:.3f}')
```

```
250/250              0s 270us/step -
loss: 629.5778 - mae: 8.0069
Test MAE: 8.056
```

0.5.2 Saving the model

We can observe that our current model has ean absolute error of 8.0069 on the test set. In other words, given a vector space of parameters within the bounds previously defined, we can predict the number of of each species with a precision that deviates from the real value by 8.0069 on average.

[ ]:
```
# save the model
model.save('bisse_model.keras')
```