opensoundscape

Release 0.4.1

Jul 24, 2020

Getting Started

OpenSoundsoundscape is free and open source software for the analysis of bioacoustic recordings. Its main goals are to allow users to train their own custom species classification models using a variety of frameworks (including convolutional neural networks) and to use trained models to predict whether species are present in field recordings. OpSo can be installed and run on a single computer or in a cluster or cloud environment.

OpenSoundcape is developed and maintained by the Kitzes Lab at the University of Pittsburgh.

The Getting Started section below provide guidance on installing OpSo. The Tutorials pages below are written as Jupyter Notebooks that can also be downloaded from the project repository on GitHub.

CHAPTER 1

OpenSoundscape

OpenSoundscape is a utility library for analyzing bioacoustic data. It consists of command line scripts for tasks such as preprocessing audio data, training machine learning models to classify vocalizations, estimating the spatial location of sounds, identifying which species' sounds are present in acoustic data, and more.

These utilities can be strung together to create data analysis pipelines. OpenSoundscape is designed to be run on any scale of computer: laptop, desktop, or computing cluster.

OpenSoundscape is currently in active development. If you find a bug, please submit an issue. If you have another question about OpenSoundscape, please email Sam Lapp (`sam.lapp` at `pitt.edu`) or Tessa Rhinehart (`tessa.rhinehart` at `pitt.edu`).

For examples of some of the utilities offered, please see the "Tutorials" section of the documentation. Included are instructions on how to download and use a pretrained machine learning model from our publicly available set of models. We plan to add additional tutorials soon.

CHAPTER 2

Installation

OpenSoundscape can be installed either via pip (for users) or poetry (for developers contributing to the code). Either way, Python 3.7 or higher is required.

2.1 Installation via pip (most users)

2.1.1 Just give me the pip command!

Already familiar with installing python packages via pip? The pip command to install OpenSoundscape is

2.1.2 Detailed instructions

Python 3.7 is required to run OpenSoundscape. Download it from this website.

We recommend installing OpenSoundscape in a virtual environment to prevent dependency conflicts. Below are instructions for installation with Python's included virtual environment manager, `venv`, but feel free to use another virtual environment manager (e.g. `conda`, `virtualenvwrapper`) if desired.

Run the following commands in your bash terminal:

- Check that you have installed Python 3.7._: `python3 --version`
- Change directories to where you wish to store the environment: `cd [path for environments folder]`
  - Tip: You can use this folder to store virtual environments for other projects as well, so put it somewhere that makes sense for you, e.g. in your home directory.
- Make a directory for virtual environments and `cd` into it: `mkdir .venv && cd .venv`
- Create an environment called `opensoundscape` in the directory: `python3 -m venv opensoundscape`
- **For Windows computers:** activate/use the environment: `opensoundscape\Scripts\activate.bat`

- **For Mac computers:** activate/use the environment `source opensoundscape/bin/activate`
- Install OpenSoundscape in the environment: `pip install opensoundscape==0.4.1`
- Once you are done with OpenSoundscape, deactivate the environment: `deactivate`
- To use the environment again, you will have to refer to absolute path of the virtual environments folder. For instance, if I were on a Mac and created `.venv` inside a directory `/Users/MyFiles/Code` I would activate the virtual environment using: `source /Users/MyFiles/Code/.venv/opensoundscape/bin/activate`

For some of our functions, you will need a version of `ffmpeg >= 0.4.1`. On Mac machines, `ffmpeg` can be installed via `brew`.

2.2 Installation via poetry (contributors and advanced users)

Poetry installation allows direct use of the most recent version of the code. This workflow allows advanced users to use the newest features in OpenSoundscape, and allows developers/contributors to build and test their contributions.

To install via poetry, do the following:

- Download poetry
- Download virtualenvwrapper
- Link `poetry` and `virtualenvwrapper`:
  - Figure out where the virtualenvwrapper.sh file is: `which virtualenvwrapper.sh`
  - Add the following to your `~/.bashrc` and source it. .. code-block:

```
# virtualenvwrapper + poetry
export PATH=~/.local/bin:$PATH
export WORKON_HOME=~/Library/Caches/pypoetry/virtualenvs
source [insert path to virtualenvwrapper.sh, e.g. ~/.local/bin/
↪virtualenvwrapper_lazy.sh]
```

- **Users**: clone this github repository to your machine: `git clone https://github.com/kitzeslab/opensoundscape.git`
- **Contributors**: fork this github repository and clone the fork to your machine
- Ensure you are in the top-level directory of the clone
- Switch to the development branch of OpenSoundscape: `git checkout develop`
- Build the virtual environment for opensoundscape: `poetry install`
  - If poetry install outputs the following error, make sure to download Python 3.7:
  - If you are on a Mac and poetry install fails to install `numba`, contact one of the developers for help troubleshooting your issues.
- Activate the virtual environment with the name provided at install e.g.: `workon opensoundscape-dxMTH98s-py3.7` or `poetry shell`
- Check that OpenSoundscape runs: `opensoundscape -h`
- Run tests (from the top-level directory): `poetry run pytest`
- Go back to your system's Python when you are done: `deactivate`

2.2.1 Jupyter

To use OpenSoundscape within JupyterLab, you will have to make an `ipykernel` for the OpenSoundscape virtual environment.

- Activate poetry virtual environment, e.g.: `workon opensoundscape-dxMTH98s-py3.7`
  - Use `poetry env list` if you're not sure what the name of the environment is
- Create ipython kernel: `python -m ipykernel install --user --name=[name of poetry environment] --display-name=OpenSoundscape`
- Now when you make a new document on JupyterLab, you should see a Python kernel available called OpenSoundscape.
- Contributors: if you include Jupyter's `autoreload`, any changes you make to the source code installed via poetry will be reflected whenever you run the `%autoreload` line magic in a cell:

2.2.2 Contributing to code

Make contributions by editing the code in your fork. Create branches for features using `git checkout -b feature_branch_name` and push these changes to remote using `git push -u origin feature_branch_name`. To merge a feature branch into the development branch, use the GitHub web interface to create a merge request.

When contributions in your fork are complete, open a pull request using the GitHub web interface. Before opening a PR, do the following to ensure the code is consistent with the rest of the package:

- Run tests: `poetry run pytest`
- Format the code with `black` style (from the top level of the repo): `black .`
- Additional libraries to be installed should be installed with `poetry add`, but in most cases contributors should not add libraries.

2.2.3 Contributing to documentation

Build the documentation using either poetry or sphinx-build

- With poetry: `poetry run build_docs`
- With sphinx-build: `sphinx-build doc doc/_build`

Publish the documentation with the following commands:

CHAPTER 3

Create spectrograms

This notebook demonstrates the use of two of OpenSoundscape's most basic classes: Audio and Spectrogram. For help installing OpenSoundscape, see the documentation

[1]:
```
# suppress warnings
import warnings
warnings.simplefilter('ignore')

# import Audio and Spectrogram classes from Opensoundscape
from opensoundscape.audio import Audio
from opensoundscape.spectrogram import Spectrogram
```

3.1 1. Create a spectrogram and save to file (as png image) with default parameters

this is the standard way to create spectrograms from audio

[2]:
```
#specify audio file
audio_path = '../tests/1min.wav'

#create Audio object
audio_object = Audio.from_file(audio_path)

#create Spectrogram object
spectrogram_object = Spectrogram.from_audio(audio_object)

#create image from Spectrogram object
spectrogram_image = spectrogram_object.to_image()

#save image file
image_path = './saved_spectrogram.png'
spectrogram_image.save(image_path)
```

3.1.1 equivalent one-line version:

[3]:
```
Spectrogram.from_audio(Audio.from_file('../tests/1min.wav')).to_image().save('./saved_
↪spectrogram.png')
```

the above example should be used for all image creation for model training.

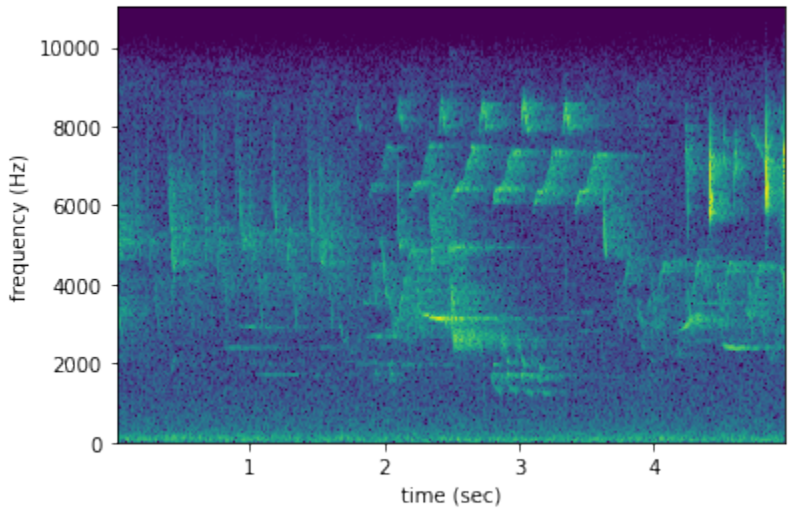
Other examples in this notebook (below) illustrate other functionality of the Spectrogram class

3.2 2. Specify target shape for the image

[4]:
```
image_shape = (224,224)
Spectrogram.from_audio(Audio.from_file('../tests/1min.wav')).to_image(shape=image_
↪shape).save('./saved_spectrogram_2.png')
```

3.3 3. Select five seconds of audio from a file, create a spectrogram, and display it

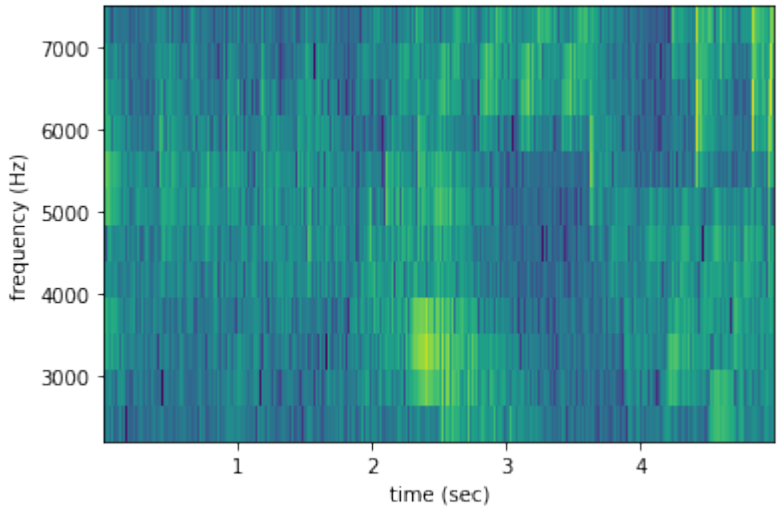
[5]:
```
#load the audio file
audio = Audio.from_file('../tests/1min.wav')

#trim to first five seconds
audio = audio.trim(0,5)

#create spectrogram
spec = Spectrogram.from_audio(audio)

#display the spectrogram
spec.plot()
```



3.4 4. Use custom parameters to create a spectrogram with high time-resolution

also trim the spectrogram in frequency and time

[6]:
```
#load audio with 44.1 kHz sampling rate
audio = Audio.from_file('../tests/1min.wav',sample_rate=44100)

#create a spectrogram with 100-sample windows (100/44100 seconds of audio per window)␣
↪and no overlap
spec = Spectrogram.from_audio(audio,window_samples=100,overlap_samples=0)

#trim the spectrogram in time and frequency
spec = spec.trim(0,5)
spec = spec.bandpass(2000,8000)
spec.plot()
```

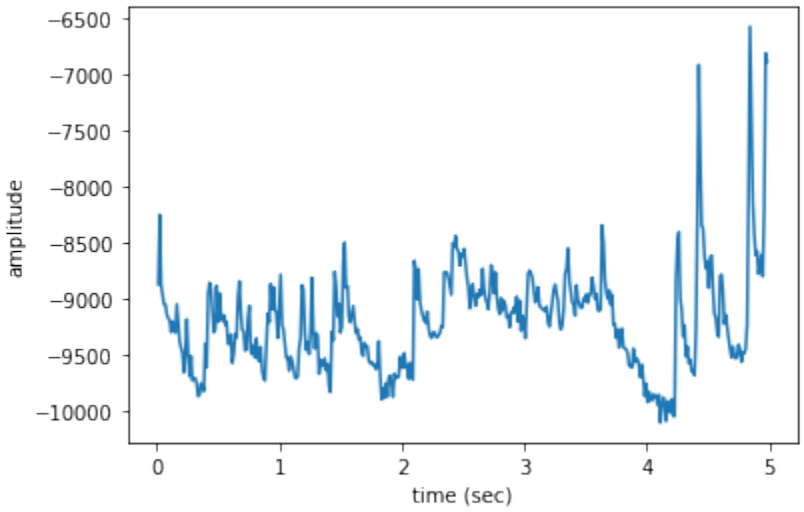


3.5 5. Create an amplitude signal from a spectrogram

[8]:
```
from matplotlib import pyplot as plt
```

[10]:
```
# make a spectrogram
spec = Spectrogram.from_audio(Audio.from_file('../tests/1min.wav').trim(0,5))

# get the amplitude signal (vertical sum) in a range of frequencies
high_freq_amplitude = spec.amplitude(freq_range=[5000,10000])
plt.plot(spec.times,high_freq_amplitude)
plt.xlabel('time (sec)')
plt.ylabel('amplitude')
```

[10]:
```
Text(0, 0.5, 'amplitude')
```



3.6 6. View the properties of a Spectrogram

[13]:
```
spec = Spectrogram.from_audio(Audio.from_file('../tests/1min.wav'))
print(f'the first few times: {spec.times[0:5]}')
print(f'the first few frequencies: {spec.frequencies[0:5]}')
```

```
the first few times: [0.01160998 0.02321995 0.03482993 0.04643991 0.05804989]
the first few frequencies: [  0.          43.06640625  86.1328125  129.19921875 172.
↪265625  ]
```

CHAPTER 4

Training a model

This quickstart will guide you through the process of creating a simple machine learning model that can identify the "peent" vocalization of an American Woodcock (*Scolopax minor*).

To use this notebook, follow the "developer" installation instructions in OpenSoundscape's README.

[1]:
```
# suppress warnings
import warnings
warnings.simplefilter('ignore')

from opensoundscape.datasets import SingleTargetAudioDataset
from opensoundscape.torch.train import train
from opensoundscape.data_selection import binary_train_valid_split
from opensoundscape.helpers import run_command
```

[2]:
```
import torch
import torch.nn
import torch.optim
import torchvision.models
```

[3]:
```
import yaml
import os.path
import pandas as pd
from pathlib import Path
from math import floor
```

4.1 Download labeled audio files

The Kitzes Lab has created some labeled ARU data of American Woodcock vocalizations. Run the following cell to download this small dataset.

These commands require you to have `wget` and `tar` installed on your computer, as they will download and unzip a compressed file in `.tar.gz` format. If you would prefer, you can also download a `.zip` version of the files by

clicking here. You will have to unzip this folder and place it in the same folder that this notebook is in.

The folder's name is `woodcock_labeled_data`.

[4]:
```
commands = [
    "curl -L https://pitt.box.com/shared/static/79fi7d715dulcldsy6uogz02rsn5uesd.gz -
↪o ./woodcock_labeled_data.tar.gz",
    "tar -xzf woodcock_labeled_data.tar.gz", # Unzip the downloaded tar.gz file
    "rm woodcock_labeled_data.tar.gz" # Remove the file after its contents are␣
↪unzipped
]
for command in commands:
    run_command(command)
```

[5]:
```
%%bash
curl -L https://pitt.box.com/shared/static/79fi7d715dulcldsy6uogz02rsn5uesd.gz -o ./
↪woodcock_labeled_data.tar.gz
tar -xzf woodcock_labeled_data.tar.gz # Unzip the downloaded tar.gz file
rm woodcock_labeled_data.tar.gz # Remove the file after its contents are unzipped
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100     7    0     7    0     0      6      0 --:--:--  0:00:01 --:--:--     6
100 4031k  100 4031k    0     0  1626k      0  0:00:02  0:00:02 --:--:-- 3296k
```

The folder contains 2s long clips. It also contains a file `woodcock_labels.csv` which contains the names of each file and its corresponding label information, created using a program called Specky.

Create a pandas DataFrame of all of the labeled files, then inspect the `head()` of this dataframe to see what its contents look like.

[6]:
```
labels = pd.read_csv(Path("woodcock_labeled_data/woodcock_labels.csv"))
labels.head()
```

[6]:

	filename	woodcock	sound_type
0	d4c40b6066b489518f8da83af1ee4984.wav	present	song
1	e84a4b60a4f2d049d73162ee99a7ead8.wav	absent	na
2	79678c979ebb880d5ed6d56f26ba69ff.wav	present	song
3	49890077267b569e142440fa39b3041c.wav	present	song
4	0c453a87185d8c7ce05c5c5ac5d525dc.wav	present	song

So that the machine learning algorithm can find these files, add the name of the folder in front of the files.

[7]:
```
labels['filename'] = 'woodcock_labeled_data' + os.path.sep + labels['filename'].
↪astype(str)
labels.head()
```

[7]:

	filename	woodcock	sound_type
0	woodcock_labeled_data/d4c40b6066b489518f8da83a...	present	song
1	woodcock_labeled_data/e84a4b60a4f2d049d73162ee...	absent	na
2	woodcock_labeled_data/79678c979ebb880d5ed6d56f...	present	song
3	woodcock_labeled_data/49890077267b569e142440fa...	present	song
4	woodcock_labeled_data/0c453a87185d8c7ce05c5c5a...	present	song

4.2 Create training and validation datasets

To use machine learning on these files, separate them into a "training" dataset, which will be used to teach the machine learning algorithm, and a "validation" dataset, which will be used to evaluate the algorithm's performance each epoch.

The "present" labels in the `woodcock` column of the dataframe will be turned into 1s. All other labels will be turned into 0s. This is required by Pytorch, which doesn't accept string labels.

[8]:
```
train_df, valid_df = binary_train_valid_split(input_df = labels, label_column=
↪'woodcock', label="present")
```

Create a list of labels so future users of the model will be able to interpret the 0/1 output.

[9]:
```
label_dict = {0:'absent', 1:'scolopax-minor'}
```

Turn these dataframes into "Datasets" using the `SingleTargetAudioDataset` class. We have to specify the names of the columns in the dataframes to use this class. Once they are set up in this class, they can be used by the training algorithm. Data augmentation could be applied in this step, but is not demonstrated here.

[10]:
```
train_dataset = SingleTargetAudioDataset(
    df=train_df, label_dict=label_dict, label_column='NumericLabels', filename_column=
↪'filename')
valid_dataset = SingleTargetAudioDataset(
    df=valid_df, label_dict=label_dict, label_column='NumericLabels', filename_column=
↪'filename')
```

4.3 Train the machine learning model

Next, we will set up the architecture of our model and train it. The model architecture we will use is a combination of a feature extractor and a classifier.

The feature extractor is a `resnet18` convolutional neural network. We call it with `pretrained=True`, so that we use a version of the model that somebody has already trained on another image dataset called ImageNet. Although spectrograms aren't the same type of images as the photographs used in ImageNet, using the pretrained model will allow the model to more quickly adapt to identifying spectrograms.

The classifier is a `Linear` classifier. We have to set the input and output size for this classifier. It takes in the outputs of the feature extractor, so `in_features = model.fc.in_features`. The model identifies one species, so it has to be able to output a "present" or "absent" classification. Thus, `out_features=2`. A multi-species model would use `out_features=number_of_species`.

[11]:
```
# Set up architecture for the type of model we will use
model = torchvision.models.resnet18(pretrained = True)
model.fc = torch.nn.Linear(in_features = model.fc.in_features, out_features = 2)
```

Next, we set up a directory in which to save results, and then run the model. We set up the following parameters: * `save_dir`: the directory in which to save results (which is created if it doesn't exist) * `model`: the model set up in the previous cell * `train_dataset`: the training dataset created using `SingleTargetAudioDataset` * `optimizer`: the optimizer to use for training the algorithm * `loss_fn`: the loss function used to assess the algorithm's performance during training * `epochs`: the number of times the model will run through the training data * `log_every`: how frequently to save performance data and save intermediate machine learning weights (`log_every=1` will save every epoch)

This function allows you to control more parameters, but they are not demonstrated here.

[12]:
```
results_path = Path('model_train_results')
if not results_path.exists(): results_path.mkdir()
train(
    save_dir = results_path,
    model = model,
    train_dataset = train_dataset,
    valid_dataset = valid_dataset,
    optimizer = torch.optim.SGD(model.parameters(), lr=1e-3),
    loss_fn = torch.nn.CrossEntropyLoss(),
    epochs=1,
    log_every=1,
    print_logging=True,
)
```

```
Epoch 0
  Training.
  Validating.
  Validation results:
    train_loss: 0.6503365182063796
    train_accuracy: 0.7272727272727273
    train_precision: [0.         0.72727273]
    train_recall: [0.         0.72727273]
    train_f1: [0.         0.72727273]
    valid_accuracy: 0.7142857142857143
    valid_precision: [0.         0.71428571]
    valid_recall: [0.         0.71428571]
    valid_f1: [0.         0.71428571]
  Saved results to model_train_results/epoch-0.tar.
Training complete.
```

This command "cleans up" by deleting all the downloaded files and results.

[13]:
```
import shutil
# Delete downloads
shutil.rmtree(Path("woodcock_labeled_data"))
# Delete results
shutil.rmtree(results_path)
```

CHAPTER 5

Model prediction

This notebook downloads an example baseline model. All baseline models are available here although they are beta models and not recommended for research use.

[1]:
```
# suppress warnings
import warnings
warnings.simplefilter('ignore')

#import modules from Opensoundscape
from opensoundscape.torch.predict import predict
from opensoundscape.datasets import SingleTargetAudioDataset
from opensoundscape.helpers import run_command
from opensoundscape.datasets import SplitterDataset
from opensoundscape.raven import lowercase_annotations
```

[2]:
```
import torch
import torch.nn
import torchvision.models
import torch.utils.data
```

[3]:
```
import yaml
import os.path
import pandas as pd
from pathlib import Path
from math import floor
```

5.1 Prepare model

5.1.1 Download model

Download the example model for Wood Thrush, *Hylocichla mustelina*.

[4]:
```
def download_from_box(link, name):
    run_command(f"curl -L {link} -o ./{name}")
```

[5]:
```
folder_name = "prediction_example"
folder_path = Path(folder_name)
if not folder_path.exists(): folder_path.mkdir()
model_filename = folder_path.joinpath("hylocichla-mustelina-epoch-4.model")
download_from_box(
    link = "https://pitt.box.com/shared/static/dslgslmag7y8ojqxv28mwhbnt7irpgeo.model
↪",
    name = model_filename
)
```

5.1.2 Load model

The model must be loaded with the same specifications that it was created with: a combination of a `resnet18` convolutional neural network and a `Linear` classifier. This model predicts two "classes": the presence and absence of Wood Thrush.

[6]:
```
num_classes = 2
model = torchvision.models.resnet18(pretrained=False)
model.fc = torch.nn.Linear(model.fc.in_features, num_classes)
model.load_state_dict(torch.load(model_filename))
#model.load_state_dict(torch.load("scolopax-minor-epoch-4.model"))
```

[6]:
```
<All keys matched successfully>
```

5.2 Prepare prediction files

Download an example soundscape which contains Wood Thrush vocalizations.

5.2.1 Download data

[7]:
```
data_filename = folder_path.joinpath("1min.wav")
download_from_box(
    link = "https://pitt.box.com/shared/static/z73eked7quh1t2pp93axzrrpq6wwydx0.wav",
    name = data_filename
)
```

5.2.2 Split data

The example soundscape must be split up into soundscapes of the same size as the ones the model was trained on. In this case, the soundscapes should be 5s long.

[8]:
```
files_to_split = [data_filename]
split_directory = folder_path.joinpath("split_files")
if not split_directory.exists(): split_directory.mkdir()
dataset = SplitterDataset(
    files_to_split,
```

(continues on next page)

(continued from previous page)

```
    overlap=0,
    duration=5,
    output_directory=split_directory,
    include_last_segment=True
)

dataloader = torch.utils.data.DataLoader(
    dataset,
    batch_size=1,
    shuffle=False,
    collate_fn=SplitterDataset.collate_fn,
)

results_csv = folder_path.joinpath("prediction_files.csv")
with open(results_csv, "w") as f:
    if False:
        f.write("Source,Annotations,Begin (s),End (s),Destination,Labels\n")
    else:
        f.write("Source,Begin (s),End (s),Destination\n")
    for idx, data in enumerate(dataloader):
        for output in data:
            f.write(f"{output}\n")
```

5.2.3 Create a Dataset

Create a dataset from these data. We create a dictionary that associates numeric labels with the class names: 1 is for predicting a Wood Thrush's presence; 0 is for predicting a Wood Thrush's absence.

[9]:
```
files_to_analyze=list(split_directory.glob("*.wav"))
sample_df = pd.DataFrame(columns=['file'],data=files_to_analyze)
```

[10]:
```
label_dict = {0:'absent', 1:'hylocichla-mustelina'}
test_dataset = SingleTargetAudioDataset(
    sample_df,
    filename_column = "file",
    label_dict = label_dict
)
```

5.3 Use model on prediction files

[11]:
```
model.eval()
prediction_df = predict(model, test_dataset, label_dict=label_dict)
prediction_df
```

[11]:

	absent	\
prediction_example/split_files/bc645003351149f4...	0.816133	
prediction_example/split_files/e36a0f200cdf42a2...	1.480433	
prediction_example/split_files/4940c91a18374102...	1.940377	
prediction_example/split_files/cfc05bd9e1b97eeb...	2.629047	
prediction_example/split_files/32747f95e81ee34c...	2.513747	
prediction_example/split_files/369134205221b5a2...	2.351259	
prediction_example/split_files/f3d6aeabe7725f64...	1.570931	

(continues on next page)

(continued from previous page)

```
prediction_example/split_files/54534197c0768b6b...  1.744635
prediction_example/split_files/e0c2d4aed1d79d4a...  1.315882
prediction_example/split_files/9d276a5dd54b631c...  1.766514
prediction_example/split_files/636f23557581b700...  0.273381
prediction_example/split_files/e55ba1b5a1316fcd...  2.138355

                                                    hylocichla-mustelina
prediction_example/split_files/bc645003351149f4...             -0.903320
prediction_example/split_files/e36a0f200cdf42a2...             -0.927409
prediction_example/split_files/4940c91a18374102...             -1.725088
prediction_example/split_files/cfc05bd9e1b97eeb...             -1.988923
prediction_example/split_files/32747f95e81ee34c...             -2.366485
prediction_example/split_files/369134205221b5a2...             -1.628652
prediction_example/split_files/f3d6aeabe7725f64...             -1.124706
prediction_example/split_files/54534197c0768b6b...             -1.055664
prediction_example/split_files/e0c2d4aed1d79d4a...             -1.407135
prediction_example/split_files/9d276a5dd54b631c...             -1.096341
prediction_example/split_files/636f23557581b700...             -0.397208
prediction_example/split_files/e55ba1b5a1316fcd...             -1.632506
```

This command "cleans up" by deleting all the downloaded files and results.

[12]:
```
import shutil
shutil.rmtree(folder_path)
```

CHAPTER 6

RIBBIT Pulse Rate model demonstration

RIBBIT (Repeat-Interval Based Bioacoustic Identification Tool) is a tool for detecting vocalizations that have a repeating structure.

This tool is useful for detecting vocalizations of frogs, toads, and other animals that produce vocalizations with a periodic structure. In this notebook, we demonstrate how to select model parameters for the Great Plains Toad, then run the model on data to detect vocalizations.

RIBBIT was introduced in the 2020 poster "Automatic Detection of Pulsed Vocalizations"

This notebook demonstrates how to use the RIBBIT tool implemented in opensoundscape as `opensoundscape.ribbit.ribbit()`

For help instaling OpenSoundscape, see the documentation

6.1 import packages

[1]:
```
# suppress warnings
import warnings
warnings.simplefilter('ignore')

#import packages
import numpy as np
from glob import glob
import pandas as pd
from matplotlib import pyplot as plt

#local imports from opensoundscape
from opensoundscape.audio import Audio
from opensoundscape.spectrogram import Spectrogram
from opensoundscape.ribbit import ribbit

# create big visuals
plt.rcParams['figure.figsize']=[15,8]
```

6.2 download example audio

first, let's download some example audio to work with.

You can run the cell below, **OR** visit this link to downlaod the data (whichever you find easier):

https://pitt.box.com/shared/static/0xclmulc4gy0obewtzbzyfnsczwgr9we.zip

If you download using the link above, first un-zip the folder (double-click on mac or right-click -> extract all on Windows). Then, move the `great_plains_toad_dataset` folder to the same location on your computer as this notebook. Then you can skip this cell:

[2]:
```
from opensoundscape.helpers import run_command
#download files from box.com to the current directory
_ = run_command(f"curl -L https://pitt.box.com/shared/static/
↪9mrxib85y1jmf1ybbjvbr0tv171iekvy.gz -o ./great_plains_toad_dataset.tar.gz")# | tar -
↪xz -f")
_ = run_command(f"tar -xz -f great_plains_toad_dataset.tar.gz")

#this will print `0` if everything went correctly. If it prints 256 or another number,
↪ something is wrong (try downloading from the link above)
```

now, you should have a folder in the same location as this notebook called `great_plains_toad_dataset`

if you had trouble accessing the data, you can try using your own audio files - just put them in a folder called `great_plains_toad_dataset` in the same location as this notebook, and this notebook will load whatever is in that folder

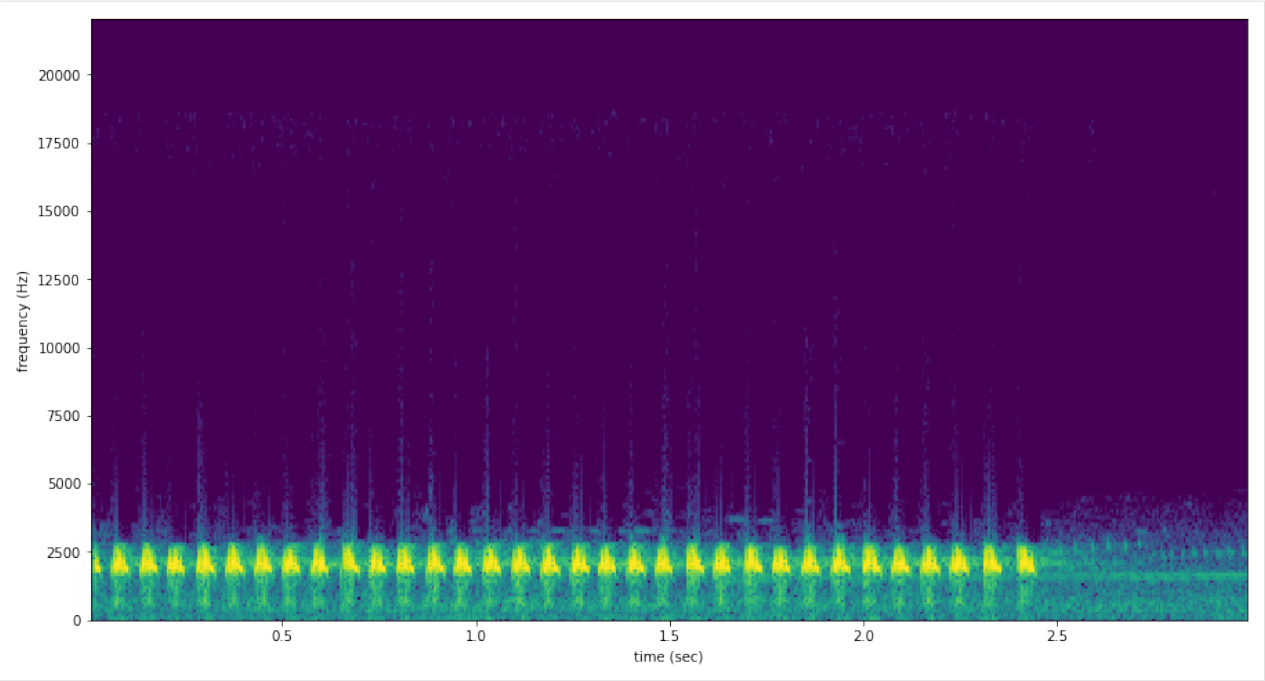
6.2.1 load an audio file and create a spectrogram

[3]:
```
audio_path = np.sort(glob('./great_plains_toad_dataset/*'))[0]

#load the audio file into an OpenSoundscape Audio object
audio = Audio.from_file(audio_path)

#trim the audio to the time from 0-3 seconds for a closer look
audio = audio.trim(0,3)

#create a Spectrogram object
spectrogram = Spectrogram.from_audio(audio)
```

6.2.2 show the Great Plains Toad spectrogram as an image

a spectrogram is a visual representation of audio with frequency on the vertical axis, time on the horizontal axis, and intensity represented by the color of the pixels

[4]:
```
spectrogram.plot()
```



6.3 select model parameters

RIBBIT requires the user to select a set of parameters that describe the target vocalization. Here is some detailed advice on how to use these parameters.

Signal Band: The signal band is the frequency range where RIBBIT looks for the target species. Based on the spectrogram above, we can see that the Great Plains Toad vocalization has the strongest energy around 2000-2500 Hz, so we will specify `signal_band = [2000,2500]`. It is best to pick a narrow signal band if possible, so that the model focuses on a specific part of the spectrogram and has less potential to include erronious sounds.

Noise Bands: Optionally, users can specify other frequency ranges called noise bands. Sounds in the `noise_bands` are *subtracted* from the `signal_band`. Noise bands help the model filter out erronious sounds from the recordings, which could include confusion species, background noise, and popping/clicking of the microphone due to rain, wind, or digital errors. It's usually good to include one noise band for very low frequencies – this specifically eliminates popping and clicking from being registered as a vocalization. It's also good to specify noise bands that target confusion species. Another approach is to specify two narrow `noise_bands` that are directly above and below the `signal_band`.

Pulse Rate Range: This parameters specifies the minimum and maximum pulse rate (the number of pulses per second, also known as pulse repetition rate) RIBBIT should look for to find the focal species. Looking at the spectrogram above, we can see that the pulse rate of this Great Plains Toad vocalization is about 15 pulses per second. By looking at other vocalizations in different environmental conditions, we notice that the pulse rate can be as slow as 10 pulses per second or as fast as 20. So, we choose `pulse_rate_range = [10, 20]` meaning that RIBBIT should look for pulses no slower than 10 pulses per second and no faster than 20 pulses per second.

Window Length: This parameter tells the algorithm how many seconds of audio to analyze at one time. Generally, you should choose a `window_length` that is similar to the length of the target species vocalization, or a little bit longer. For very slowly pulsing vocalizations, choose a longer window so that at least 5 pulses can occur in one window (0.5 pulses per second -> 10 second window). Typical values for `window_length` are 1 to 10 seconds. Keep in mind that The Great Plains Toad has a vocalization that continues on for many seconds (or minutes!) so we chose a 2-second window which will include plenty of pulses.

Plot: We can choose to show the power spectrum of pulse repetition rate for each window by setting `plot=True`. The default is not to show these plots (`plot=False`).

[5]:
```
# minimum and maximum rate of pulsing (pulses per second) to search for
pulse_rate_range = [10,20]

# look for a vocalization in the range of 1000-2000 Hz
signal_band = [2000,2500]

# subtract the amplitude signal from these frequency ranges
noise_bands = [ [0,200], [10000,10100]]

#divides the signal into segments this many seconds long, analyzes each independently
window_length = 2 #(seconds)

#if True, it will show the power spectrum plot for each audio segment
show_plots = True
```

6.4 search for pulsing vocalizations with `ribbit()`

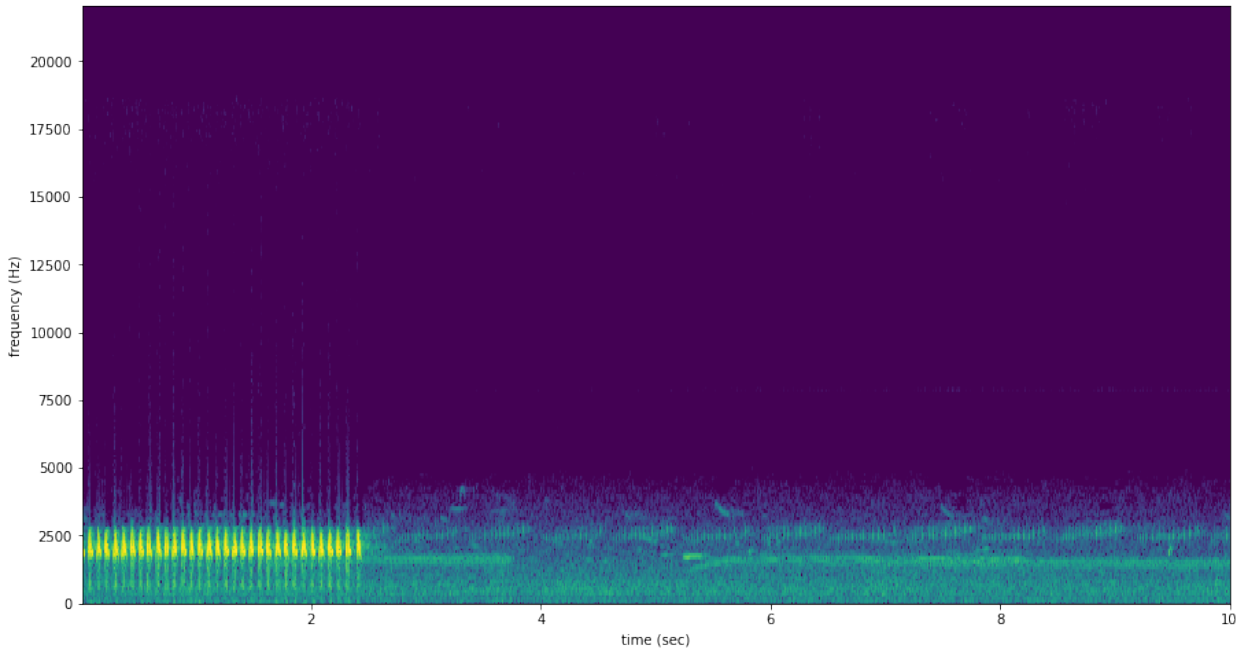
This function takes the parameters we chose above as arguments, performs the analysis, and returns two arrays: - **scores:** the pulse rate score for each window - **times:** the start time in seconds of each window

The scores output by the function may be very low or very high. They do not represent a "confidence" or "probability" from 0 to 1. Instead, the relative values of scores on a set of files should be considered: when RIBBIT detects the target species, the scores will be significantly higher than when the species is not detected.

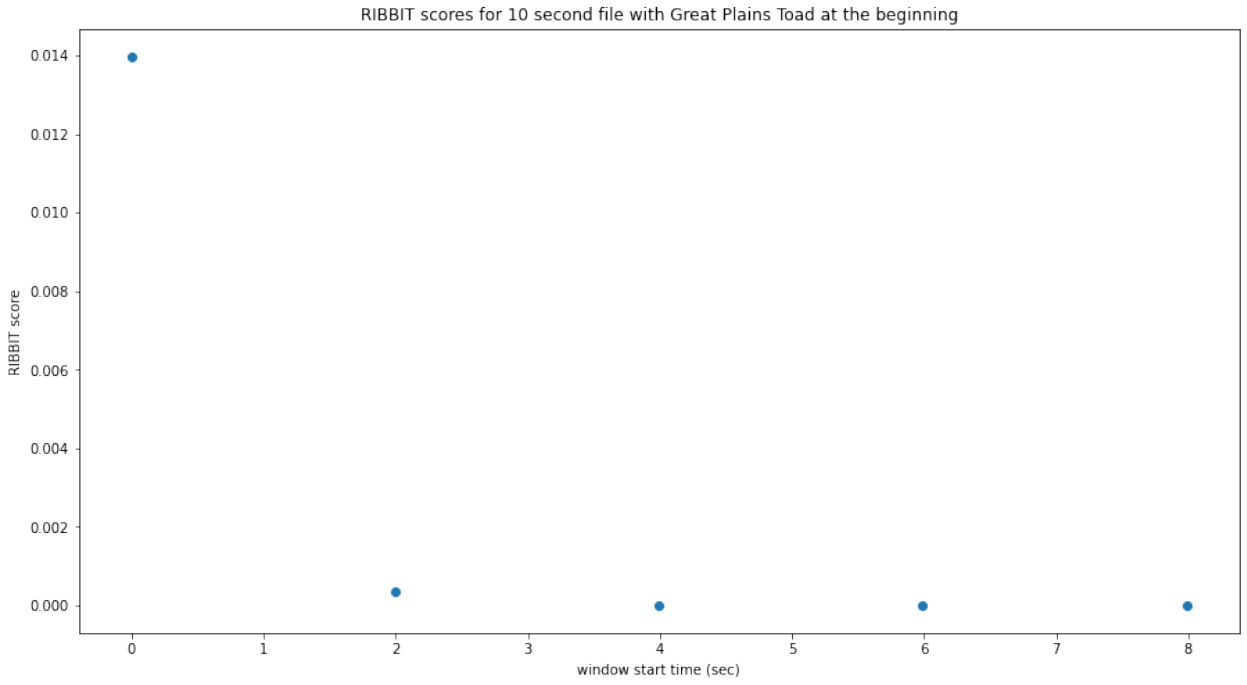
The file `gpt0.wav` has a Great Plains Toad vocalizing only at the beginning. Let's analyze the file with RIBBIT and look at the scores versus time.

[8]:
```
#get the audio file path
audio_path = np.sort(glob('./great_plains_toad_dataset/*'))[0]

#make the spectrogram
spec = Spectrogram.from_audio(audio.from_file(audio_path))

#run RIBBIT
scores, times =  ribbit(
                    spec,
                    pulse_rate_range=pulse_rate_range,
                    signal_band=signal_band,
                    window_len=window_length,
                    noise_bands=noise_bands,
                    plot=False)


#show the spectrogram
print('spectrogram of 10 second file with Great Plains Toad at the beginning')
spec.plot()

# plot the score vs time of each window
plt.scatter(times,scores)
plt.xlabel('window start time (sec)')
plt.ylabel('RIBBIT score')
plt.title('RIBBIT scores for 10 second file with Great Plains Toad at the beginning')
```

```
spectrogram of 10 second file with Great Plains Toad at the beginning
```




```
[8]: Text(0.5, 1.0, 'RIBBIT scores for 10 second file with Great Plains Toad at the␣
     ↪beginning')
```



as we hoped, RIBBIT outputs a high score during the vocalization (the window from 0-2 seconds) and a low score when the frog is not vocalizing

6.5 analyzing a set of files

[12]:
```
# set up a dataframe for storing files' scores and labels
df = pd.DataFrame(index = glob('./great_plains_toad_dataset/*'),columns=['score',
↪'label'])

# label is 1 if the file contains a Great Plains Toad vocalization, and 0 if it does␣
↪not
df['label'] = [1 if 'gpt' in f else 0 for f in df.index]

# calculate RIBBIT scores
for path in df.index:

    #make the spectrogram
    spec = Spectrogram.from_audio(audio.from_file(path))

    #run RIBBIT
    scores, times =  ribbit(
                        spec,
                        pulse_rate_range=pulse_rate_range,
                        signal_band=signal_band,
                        window_len=window_length,
                        noise_bands=noise_bands,
                        plot=False)

    # use the maximum RIBBIT score from any window as the score for this file
    # multiply the score by 10,000 to make it easier to read
    df.at[path,'score'] = max(scores) * 10000

print("Files sorted by score, from highest to lowest:")
df.sort_values(by='score',ascending=False)
```

```
Files sorted by score, from highest to lowest:
```

[12]:

	score	label
./great_plains_toad_dataset/gpt0.mp3	139.765	1
./great_plains_toad_dataset/gpt3.mp3	13.8338	1
./great_plains_toad_dataset/gpt2.mp3	8.25766	1
./great_plains_toad_dataset/gpt1.mp3	6.1136	1
./great_plains_toad_dataset/negative3.mp3	2.34044	0
./great_plains_toad_dataset/negative2.mp3	1.73015	0
./great_plains_toad_dataset/negative4.mp3	1.56953	0
./great_plains_toad_dataset/negative1.mp3	1.21802	0
./great_plains_toad_dataset/negative9.mp3	1.13301	0
./great_plains_toad_dataset/negative8.mp3	1.08165	0
./great_plains_toad_dataset/negative6.mp3	0.966176	0
./great_plains_toad_dataset/negative5.mp3	0.695368	0
./great_plains_toad_dataset/gpt4.mp3	0.634423	1
./great_plains_toad_dataset/pops2.mp3	0.51215	0
./great_plains_toad_dataset/water.mp3	0.510283	0
./great_plains_toad_dataset/pops1.mp3	0.493911	0
./great_plains_toad_dataset/negative7.mp3	0.0156971	0
./great_plains_toad_dataset/silent.mp3	0	0

So, how good is RIBBIT at finding the Great Plains Toad?

We can see that the scores for all of the files with Great Plains Toad (gpt) score above 6 except `gpt4.mp3` (which contains only a very quiet and distant vocalization). All files that do not contain the Great Plains Toad score less than 2.5. So, RIBBIT is doing a good job separating Great Plains Toads vocalizations from other sounds!

Notably, noisy files like `pops1.mp3` score low even though they have lots of periodic energy - our `noise_bands` sucessfully rejected these files. Without using `noise_bands`, files like these would receive very high scores. Also, some birds in "negatives" files that have periodic calls around the same pulsre rate as the Great Plains Toad received low scores. This is also a result of choosing a tight `signal_band` and strategic `noise_bands`. You can try adjusting or eliminating these bands to see their effect on the audio.

(HINT: elimintating the `noise_bands` will result in high scores for the "pops" files)

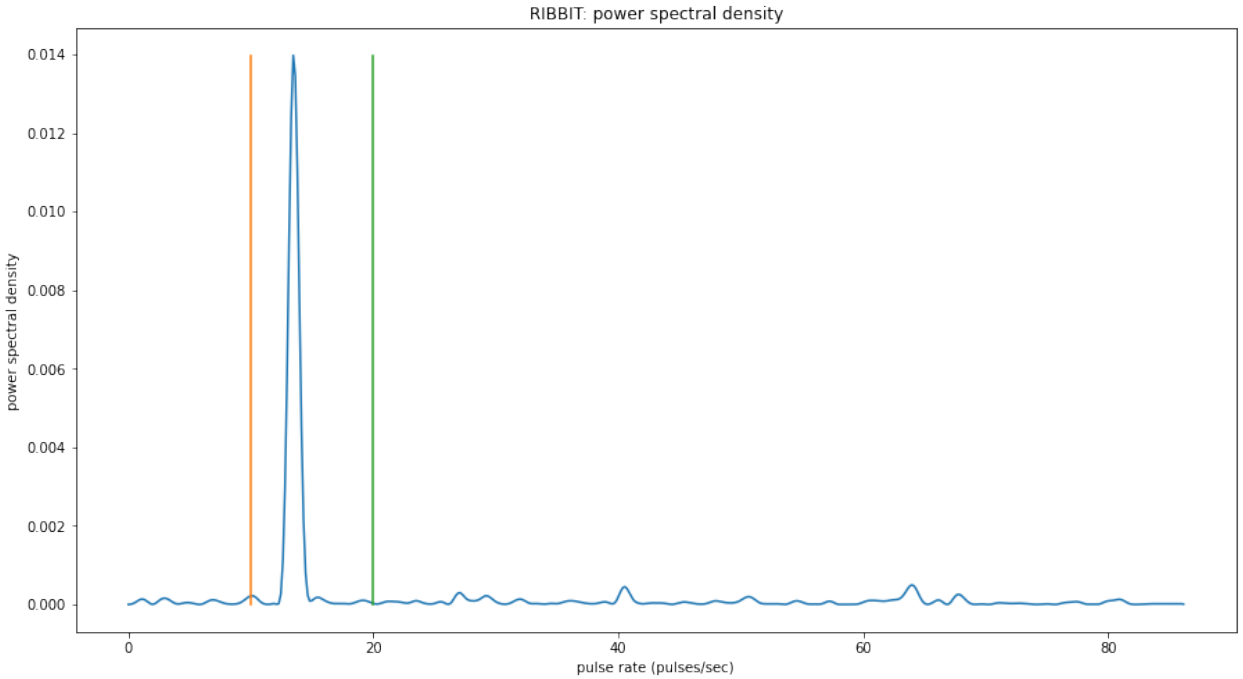
6.6 detail view

Now, lets look at one 10 second file and tell ribbit to plot the power spectral density for each window (`plot=True`). This way, we can see if peaks are emerging at the expected pulse rates. Since our `window_length` is 2 seconds, each of these plots represents 2 seconds of audio. The vertical lines on the power spectral density represent the lower and upper `pulse_rate_range` limits.

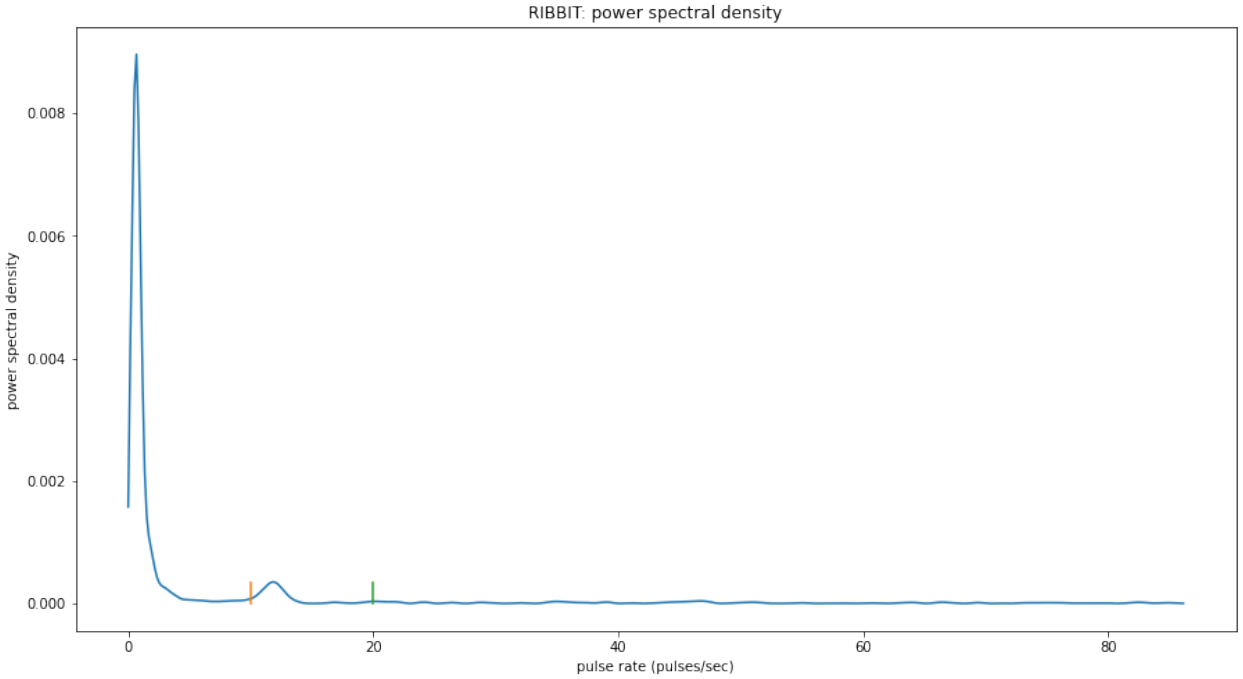
In the file `gpt0.mp3`, the Great Plains Toad vocalizes for a couple seconds at the beginning, then stops. We expect to see a peak in the power spectral density at 15 pulses/sec in the first 2 second window, and maybe a bit in the second, but not later in the audio.

[45]:
```
#create a spectrogram from the file, like above:
# 1. get audio file path
audio_path = np.sort(glob('./great_plains_toad_dataset/*'))[0]
# 2. make audio object and trim (this time 0-10 seconds)
audio = Audio.from_file(audio_path).trim(0,10)
# 3. make spectrogram
spectrogram = Spectrogram.from_audio(audio)

scores, times =  ribbit(
                        spectrogram,
                        pulse_rate_range=pulse_rate_range,
                        signal_band=signal_band,
                        window_len=window_length,
                        noise_bands=noise_bands,
                        plot=show_plots)
```

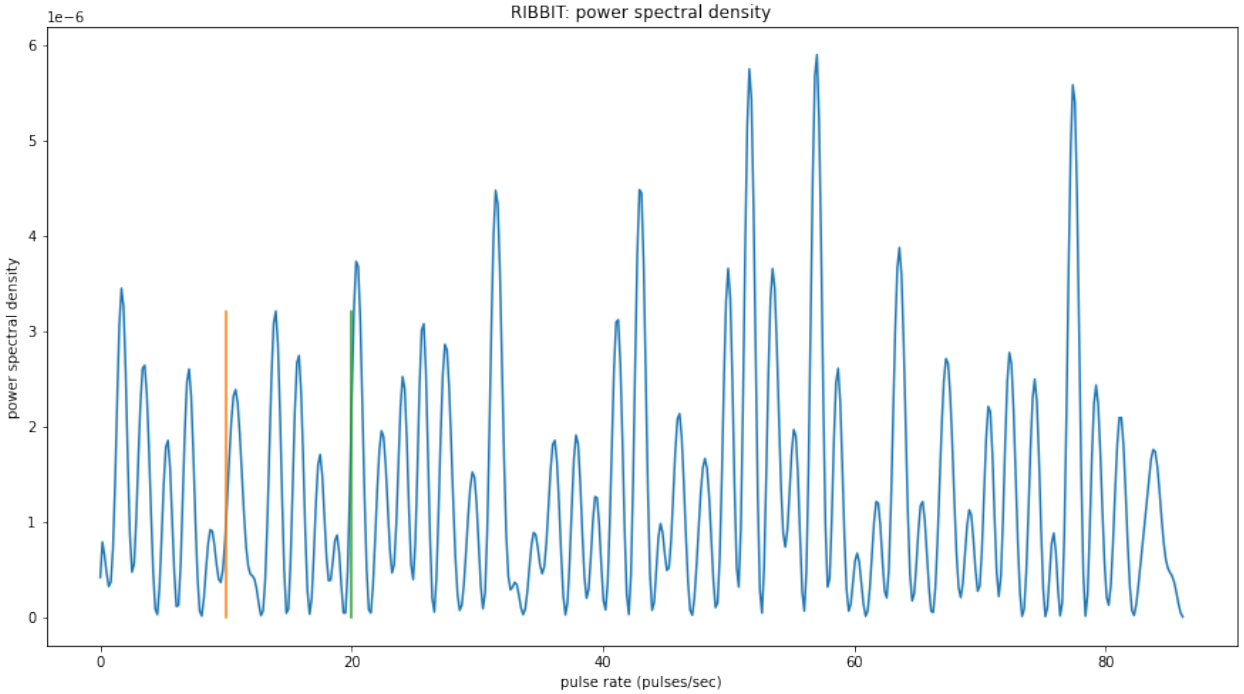
```
window: 0.0000 sec to 1.9969 sec
peak freq: 13.4583
```



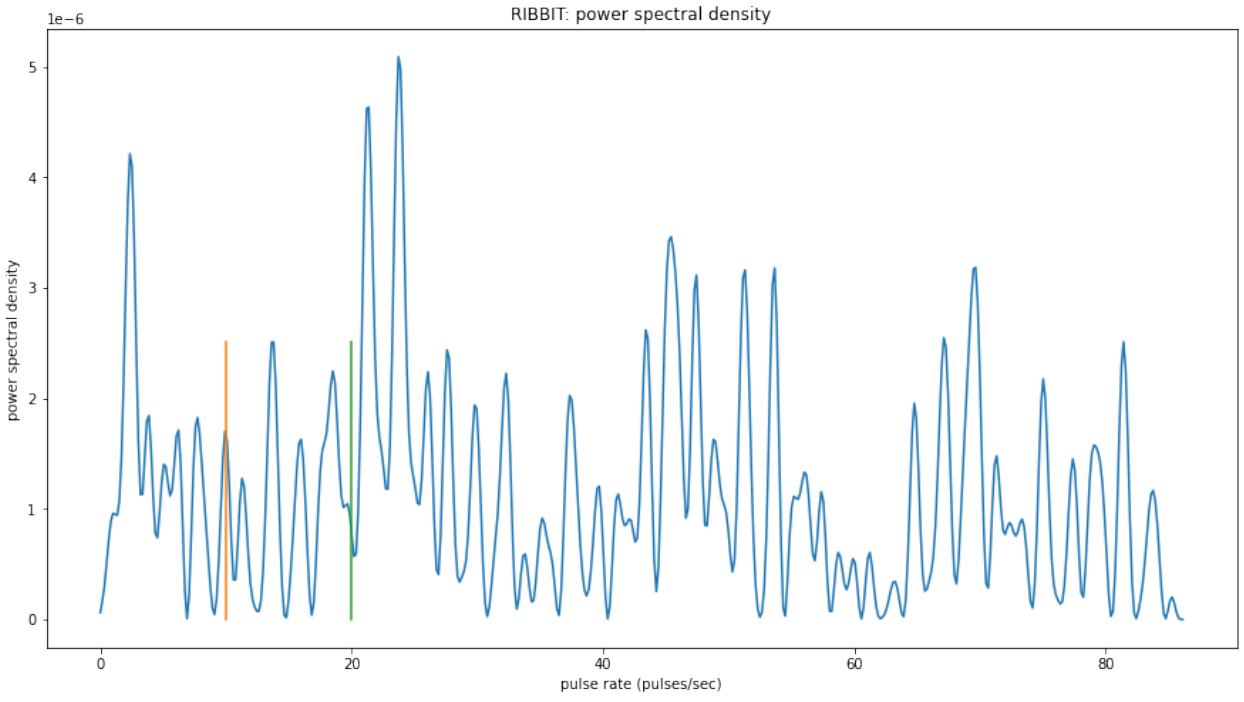

```
window: 1.9969 sec to 3.9938 sec
peak freq: 0.6729
```




```
window: 3.9938 sec to 5.9907 sec
peak freq: 57.0293
```




```
window: 5.9907 sec to 7.9877 sec
peak freq: 23.7202
```



6.7 Time to experiment for yourself

Now that you know the basics of how to use RIBBIT, you can try using it on your own data. We recommend spending some time looking at different recordings of your focal species before choosing parameters. Experiment with the noise bands and window length, and get in touch if you have questions!

Sam's email: sam . lapp [at] pitt.edu

this cell will delete the folder `great_plains_toad_dataset`. Only run it if you wish delete that folder and the example audio inside it.

[3]:
```
_ = run_command('rm -r ./great_plains_toad_dataset/')
_ = run_command('rm ./great_plains_toad_dataset.tar.gz')
```

[ ]:
```

```

CHAPTER 7

API Documentation

7.1 Audio

audio.py: Utilities for dealing with audio files

class opensoundscape.audio.**Audio**(*samples*, *sample_rate*)
Container for audio samples

bandpass(*low_f*, *high_f*, *order=9*)
bandpass audio signal frequencies

uses a phase-preserving algorithm (scipy.signal's butter and solfiltfilt)

Parameters

- **low_f** – low frequency cutoff (-3 dB) in Hz of bandpass filter
- **high_f** – high frequency cutoff (-3 dB) in Hz of bandpass filter
- **order** – butterworth filter order (integer) ~= steepness of cutoff

duration()
Return duration of Audio

Output: duration (float): The duration of the Audio

classmethod from_bytesio(*bytesio*, *sample_rate=None*, *resample_type='kaiser_fast'*)
...

classmethod from_file(*path*, *sample_rate=None*, *max_duration=None*, *resample_type='kaiser_fast'*)
Load audio from files

Deal with the various possible input types to load an audio file and generate a spectrogram

Parameters

- **path** (*str, Path*) – path to an audio file

- **sample_rate** (*int, None*) – resample audio with value and resample_type, if None use source sample_rate (default: None)
- **resample_type** – method used to resample_type (default: kaiser_fast)
- **max_duration** – the maximum length of an input file, None is no maximum (default: None)

Returns attributes samples and sample_rate

Return type *Audio*

save(*path*)
save Audio to file

Parameters **path** – destination for output

spectrum()
create frequency spectrum from an Audio object using fft

Parameters **self** –

Returns fft, frequencies

time_to_sample(*time*)
Given a time, convert it to the corresponding sample

Parameters **time** – The time to multiply with the sample_rate

Returns The rounded sample

Return type sample

trim(*start_time*, *end_time*)
trim Audio object in time

Parameters

- **start_time** – time in seconds for start of extracted clip
- **end_time** – time in seconds for end of extracted clip

Returns a new Audio object containing samples from start_time to end_time

exception opensoundscape.audio.**OpsoLoadAudioInputError**
Custom exception indicating we can't load input

exception opensoundscape.audio.**OpsoLoadAudioInputTooLong**
Custom exception indicating length of audio is too long

7.2 Audio Tools

audio_tools.py: set of tools that filter or modify audio files or sample arrays (not Audio objects)

opensoundscape.audio_tools.**bandpass_filter**(*signal*, *low_f*, *high_f*, *sample_rate*, *order=9*)
perform a butterworth bandpass filter on a discrete time signal using scipy.signal's butter and solfiltfilt (phase-preserving version of sosfilt)

Parameters

- **signal** – discrete time signal (audio samples, list of float)
- **low_f** – -3db point (?) for highpass filter (Hz)
- **high_f** – -3db point (?) for highpass filter (Hz)

- **sample_rate** – samples per second (Hz)
- **order=9** – higher values -> steeper dropoff

Returns filtered time signal

opensoundscape.audio_tools.**butter_bandpass**(*low_f*, *high_f*, *sample_rate*, *order=9*)
generate coefficients for bandpass_filter()

Parameters

- **low_f** – low frequency of butterworth bandpass filter
- **high_f** – high frequency of butterworth bandpass filter
- **sample_rate** – audio sample rate
- **order=9** – order of butterworth filter

Returns set of coefficients used in sosfiltfilt()

opensoundscape.audio_tools.**clipping_detector**(*samples*, *threshold=0.6*)
count the number of samples above a threshold value

Parameters

- **samples** – a time series of float values
- **threshold=0.6** – minimum value of sample to count as clipping

Returns number of samples exceeding threshold

opensoundscape.audio_tools.**convolve_file**(*in_file*, *out_file*, *ir_file*, *input_gain=1.0*)
apply an impulse_response to a file using ffmpeg's afir convolution

ir_file is an audio file containing a short burst of noise recorded in a space whose acoustics are to be recreated

this makes the files 'sound as if' it were recorded in the location that the impulse response (ir_file) was recorded

Parameters

- **in_file** – path to an audio file to process
- **out_file** – path to save output to
- **ir_file** – path to impulse response file
- **input_gain=1.0** – ratio for in_file sound's amplitude in (0,1)

Returns os response of ffmpeg command

opensoundscape.audio_tools.**mixdown_with_delays**(*files_to_mix*, *destination*, *delays=None*, *levels=None*, *duration='first'*, *verbose=0*, *create_txt_file=False*)
use ffmpeg to mixdown a set of audio files, each starting at a specified time (padding beginnings with zeros)

Parameters

- **files_to_mix** – list of audio file paths
- **destination** – path to save mixdown to
- **delays=None** – list of delays (how many seconds of zero-padding to add at beginning of each file)
- **levels=None** – optionally provide a list of relative levels (amplitudes) for each input
- **duration='first'** – ffmpeg option for duration of output file: match duration of 'longest','shortest',or 'first' input file

- **verbose=0** – if >0, prints ffmpeg command and doesn't suppress ffmpeg output (command line output is returned from this function)
- **create_txt_file=False** – if True, also creates a second output file which lists all files that were included in the mixdown

Returns ffmpeg command line output

opensoundscape.audio_tools.**silence_filter**(*filename*, *smoothing_factor=10*, *window_len_samples=256*, *overlap_len_samples=128*, *threshold=None*)

Identify whether a file is silent (0) or not (1)

Load samples from an mp3 file and identify whether or not it is likely to be silent. Silence is determined by finding the energy in windowed regions of these samples, and normalizing the detected energy by the average energy level in the recording.

If any windowed region has energy above the threshold, returns a 0; else returns 1.

Parameters

- **filename** (*str*) – file to inspect
- **smoothing_factor** (*int*) – modifier to window_len_samples
- **window_len_samples** – number of samples per window segment
- **overlap_len_samples** – number of samples to overlap each window segment
- **threshold** – threshold value (experimentally determined)

Returns 0 if file contains no significant energy over bakcground 1 if file contains significant energy over bakcground

If threshold is None: returns net_energy over background noise

opensoundscape.audio_tools.**window_energy**(*samples*, *window_len_samples=256*, *overlap_len_samples=128*)

Calculate audio energy with a sliding window

Calculate the energy in an array of audio samples

Parameters

- **samples** (*np.ndarray*) – array of audio samples loaded using librosa.load
- **window_len_samples** – samples per window
- **overlap_len_samples** – number of samples shared between consecutive windows

Returns list of energy level (float) for each window

7.3 Commands

opensoundscape.commands.**run_command**(*cmd*)

Run a command returning output, error

Input: cmd: A string containing some command

Output: (stdout, stderr): A tuple of standard out and standard error

opensoundscape.commands.**run_command_return_code**(*cmd*)

Run a command returning the return code

Input: cmd: A string containing some command

Output: return_code: The return code of the function

7.4 Completions

7.5 Config

opensoundscape.config.**get_default_config**()
Get the default configuration file as a dictionary

Output: dict: A dictionary containing the default Opensoundscape configuration

opensoundscape.config.**validate**(*config*)
Validate a configuration string

Input: config: A string containing an Opensoundscape configuration

Output: dict: A dictionary of the validated Opensoundscape configuration

opensoundscape.config.**validate_file**(*fname*)
Validate a configuration file

Input: fname: A filename containing an Opensoundscape configuration

Output: dict: A dictionary of the validated Opensoundscape configuration

7.6 Console Checks

Utilities related to console checks on docopt args

7.7 Console

console.py: Entrypoint for opensoundscape

opensoundscape.console.**build_docs**()
Run sphinx-build for our project

opensoundscape.console.**entrypoint**()
The Opensoundscape entrypoint for console interaction

7.8 Data Selection

opensoundscape.data_selection.**binary_train_valid_split**(*input_df*, *label*, *label_column='Labels'*, *train_size=0.8*, *random_state=101*)
Split a dataset into train and validation dataframes

Given a Dataframe and a label in column "Labels" (singly labeled) generate a train dataset with ~80% of each label and a valid dataset with the rest.

Parameters

- **input_df** – A singly-labeled CSV file

- **label** – One of the labels in the column label_column to use as a positive label (1), all others are negative (0)
- **label_column** – Name of the column that labels should come from [default: "Labels"]
- **train_size** – The decimal fraction to use for the training set [default: 0.8]
- **random_state** – The random state to use for train_test_split [default: 101]

Output: train_df: A Dataframe containing the training set valid_df: A Dataframe containing the validation set

opensoundscape.data_selection.**expand_multi_labeled**(*input_df*)
Given a multi-labeled dataframe, generate a singly-labeled dataframe

Given a Dataframe with a "Labels" column that is multi-labeled (e.g. "hello|world") split the row into singly labeled rows.

Parameters **input_df** – A Dataframe with a multi-labeled "Labels" column (separated by "|")

Output: output_df: A Dataframe with singly-labeled "Labels" column

opensoundscape.data_selection.**upsample**(*input_df*, *label_column='Labels'*, *random_state=None*)
Given a input DataFrame upsample to maximum value

Upsampling removes the class imbalance in your dataset. Rows for each label are repeated up to *max_count // rows*. Then, we randomly sample the rows to fill up to *max_count*.

Input: input_df: A DataFrame to upsample label_column: The column to draw unique labels from random_state: Set the random_state during sampling

Output: df: An upsampled DataFrame

7.9 Datasets

class opensoundscape.datasets.**SingleTargetAudioDataset**(*df*, *label_dict*, *filename_column='Destination'*, *from_audio=True*, *label_column=None*, *height=224*, *width=224*, *add_noise=False*, *debug=None*, *random_trim_length=None*, *max_overlay_num=0*, *overlay_prob=0.2*, *overlay_weight='random'*)
Single Target Audio -> Image Dataset

Given a DataFrame with audio files in one of the columns, generate a Dataset of spectrogram images for basic machine learning tasks.

This class provides access to several types of augmentations that act on audio and images with the following arguments: - add_noise: for adding RandomAffine and ColorJitter noise to images - random_trim_length: for only using a short random clip extracted from the training data - max_overlay_num / overlay_prob / overlay_weight:

controlling the maximum number of additional spectrograms to overlay, the probability of overlaying an individual spectrogram, and the weight for the weighted sum of the spectrograms

Additional augmentations on tensors are available when calling *train()* from the module *opensoundscape.torch.train*.

Input: df: A DataFrame with a column containing audio files label_dict: a dictionary mapping numeric labels to class names,

- for example: {0:'American Robin',1:'Northern Cardinal'}
- pass *None* if you wish to retain numeric labels

filename_column: The column in the DataFrame which contains paths to data [default: Destination] from_audio: Whether the raw dataset is audio [default: True] label_column: The column with numeric labels if present [default: None] height: Height for resulting Tensor [default: 224] width: Width for resulting Tensor [default: 224] add_noise: Apply RandomAffine and ColorJitter filters [default: False] debug: Save images to a directory [default: None] random_trim_length: Extract a clip of this many seconds of audio starting at a random time

If None, the original clip will be used [default: None]

max_overlay_num: the maximum number of additional images to overlay, each with probability overlay_prob [default: 0] overlay_prob: Probability of an image from a different class being overlayed (combined as a weighted sum)

on the training image. typical values: 0, 0.66 [default: 0.2]

overlay_weight: the weight given to the overlaid image during augmentation. When 'random', will randomly select a different weight between 0.2 and 0.5 for each overlay When not 'random', should be a float between 0 and 1 [default: 'random']

Output:

Dictionary: { "X": (3, H, W) , "y": (1) if label_column != None }

image_from_audio(*audio*, *mode='RGB'*)
Create a PIL image from audio

Inputs: audio: audio object mode: PIL image mode, e.g. "L" or "RGB" [default: RGB]

overlay_random_image(*original_image*, *original_length*, *original_class*, *original_path*)
Overlay an image from another class

Select a random file from a different class. Trim if necessary to the same length as the given image. Overlay the images on top of each other with a weight

class opensoundscape.datasets.**SplitterDataset**(*wavs*, *annotations=False*, *label_corrections=None*, *overlap=1*, *duration=5*, *output_directory='segments'*, *include_last_segment=False*)
A PyTorch Dataset for splitting a WAV files

Inputs: wavs: A list of WAV files to split annotations: Should we search for corresponding annotations files? (default: False) label_corrections: Specify a correction labels CSV file w/ column headers "raw" and "corrected" (default: None) overlap: How much overlap should there be between samples (units: seconds, default: 1) duration: How long should each segment be? (units: seconds, default: 5) output_directory Where should segments be written? (default: segments/) include_last_segment: Do you want to include the last segment? (default: False)

Effects:

- Segments will be written to the output_directory

Outputs:

output: A list of CSV rows containing the source audio, segment begin time (seconds), segment end time (seconds), segment audio, and present classes separated by 'l' if annotations were requested

opensoundscape.datasets.**annotations_with_overlaps_with_clip**(*df*, *begin*, *end*)
Determine if any rows overlap with current segment

Inputs: df: A dataframe containing a Raven annotation file begin: The begin time of the current segment (unit: seconds) end: The end time of the current segment (unit: seconds)

Output: sub_df: A dataframe of annotations which overlap with the begin/end times

opensoundscape.datasets.**get_md5_digest**(*input_string*)
Generate MD5 sum for a string

Inputs: input_string: An input string

Outputs: output: A string containing the md5 hash of input string

7.10 Grad Cam

7.11 Helpers

opensoundscape.helpers.**binarize**(*x*, *threshold*)
return a list of 0, 1 by thresholding vector x

opensoundscape.helpers.**bound**(*x*, *bounds*)
restrict x to a range of bounds = [min, max]

opensoundscape.helpers.**file_name**(*path*)
get file name without extension from a path

opensoundscape.helpers.**hex_to_time**(*s*)
convert a hexidecimal, Unix time string to a datetime timestamp

opensoundscape.helpers.**isNan**(*x*)
check for nan by equating x to itself

opensoundscape.helpers.**jitter**(*x*, *width*, *distribution='gaussian'*)
Jitter (add random noise to) each value of x

Parameters

- **x** – scalar, array, or nd-array of numeric type
- **width** – multiplier for random variable (stdev for 'gaussian' or r for 'uniform')
- **distribution** – 'gaussian' (default) or 'uniform' if 'gaussian': draw jitter from gaussian with mu = 0, std = width if 'uniform': draw jitter from uniform on [-width, width]

Returns x + random jitter

Return type jittered_x

opensoundscape.helpers.**linear_scale**(*array*, *in_range=(0, 1)*, *out_range=(0, 255)*)
Translate from range in_range to out_range

Inputs: in_range: The starting range [default: (0, 1)] out_range: The output range [default: (0, 255)]

Outputs: new_array: A translated array

opensoundscape.helpers.**min_max_scale**(*array*, *feature_range=(0, 1)*)
rescale vaues in an a array linearly to feature_range

opensoundscape.helpers.**rescale_features**(*X*, *rescaling_vector=None*)
rescale all features by dividing by the max value for each feature

optionally provide the rescaling vector (1xlen(X) np.array), so that you can rescale a new dataset consistently with an old one

returns rescaled feature set and rescaling vector

opensoundscape.helpers.**run_command**(*cmd*)
run a bash command with Popen, return response

opensoundscape.helpers.**sigmoid**(*x*)
sigmoid function

7.12 Localization

opensoundscape.localization.**calc_speed_of_sound**(*temperature=20*)
Calculate speed of sound in meters per second

Calculate speed of sound for a given temperature in Celsius (Humidity has a negligible effect on speed of sound and so this functionality is not implemented)

Parameters **temperature** – ambient temperature in Celsius

Returns the speed of sound in meters per second

opensoundscape.localization.**localize**(*receiver_positions*, *arrival_times*, *temperature=20.0*, *invert_alg='gps'*, *center=True*, *pseudo=True*)
Perform TDOA localization on a sound event

Localize a sound event given relative arrival times at multiple receivers. This function implements a localization algorithm from the equations described in the class handout ("Global Positioning Systems"). Localization can be performed in a global coordinate system in meters (i.e., UTM), or relative to recorder positions in meters.

Parameters

- **receiver_positions** – a list of [x,y,z] positions for each receiver Positions should be in meters, e.g., the UTM coordinate system.
- **arrival_times** – a list of TDOA times (onset times) for each recorder The times should be in seconds.
- **temperature** – ambient temperature in Celsius
- **invert_alg** – what inversion algorithm to use
- **center** – whether to center recorders before computing localization result. Computes localization relative to centered plot, then translates solution back to original recorder locations. (For behavior of original Sound Finder, use True)
- **pseudo** – whether to use the pseudorange error (True) or sum of squares discrepancy (False) to pick the solution to return (For behavior of original Sound Finder, use False. However, in initial tests, pseudorange error appears to perform better.)

Returns The solution (x,y,z,b) with the lower sum of squares discrepancy b is the error in the pseudorange (distance to mics), b=c*delta_t (delta_t is time error)

opensoundscape.localization.**lorentz_ip**(*u*, *v=None*)
Compute Lorentz inner product of two vectors

For vectors *u* and *v*, the Lorentz inner product for 3-dimensional case is defined as

u[0]*v[0] + u[1]*v[1] + u[2]*v[2] - u[3]*v[3]

Or, for 2-dimensional case as

u[0]*v[0] + u[1]*v[1] - u[2]*v[2]

Args u: vector with shape either (3,) or (4,) v: vector with same shape as x1; if None (default), sets v = u

Returns float: value of Lorentz IP

opensoundscape.localization.**travel_time**(*source*, *receiver*, *speed_of_sound*)

Calculate time required for sound to travel from a souce to a receiver

Parameters

- **source** – cartesian position [x,y] or [x,y,z] of sound source
- **receiver** – cartesian position [x,y] or [x,y,z] of sound receiver
- **speed_of_sound** – speed of sound in m/s

Returns time in seconds for sound to travel from source to receiver

7.13 Metrics

7.14 Pulse Finder

7.15 PyTorch Prediction

DEPRECATED: use opensoundscape.torch.predict instead

these functions are currently used only to support *localization.py* the module contains a pytorch prediction function (deprecated) and some additional functionality for using gradcam

opensoundscape.pytorch_prediction.**activation_region_limits**(*gcam*, *threshold=0.2*)

calculate bounds of a GradCam activation region

Parameters

- **gcam** – a 2-d array gradcam activation array generated by gradcam_region()
- **threshold=0.2** – minimum value of gradcam (0-1) to count as 'activated'

Returns [[min row, max_row], [min_col, max_col]] indices of gradcam elements exceeding threshold

opensoundscape.pytorch_prediction.**activation_region_to_box**(*activation_region*, *threshold=0.2*)

draw a rectangle of the activation box as a boolean array (useful for plotting a mask over a spectrogram)

Parameters

- **activation_region** – a 2-d gradcam activation array
- **threshold=0.2** – minimum value of activation to count as 'activated'

Returns mask 2-d array of 0, 1 where 1's form a solid box of activated region

opensoundscape.pytorch_prediction.**gradcam_region**(*model*, *img_paths*, *img_shape*, *predictions=None*, *save_gcams=True*, *box_threshold=0.2*)

Compute the GradCam activation region (the area of an image that was most important for classification in the CNN)

Parameters

- **model** – a pytorch model object
- **img_paths** – list of paths to image files
- **= None** (*predictions*) – [list of float] optionally, provide model predictions per file to avoid re-computing
- **= True** (*save_gcams*) – bool, if False only box regions around gcams are saved

Returns limits of the box surrounding the gcam activation region, as indices: [[min row, max row], [min col, max col]] gcams: (only returned if save_gcams == True) arrays with gcam activation values, shape = shape of image

Return type boxes

opensoundscape.pytorch_prediction.**in_box**(*x*, *y*, *box_lims*)

check if an x, y position falls within a set of limits

Parameters

- **x** – first index
- **y** – second index
- **box_lims** – [[x low,x high], [y low,y high]]

Returns: True if (x,y) is in box_lims, otherwise False

opensoundscape.pytorch_prediction.**predict**(*model*, *img_paths*, *img_shape*, *batch_size=1*, *num_workers=12*, *apply_softmax=True*)

get multi-class model predictions from a pytorch model for a set of images

Parameters

- **model** – a pytorch model object (not path to weights)
- **img_paths** – a list of paths to RGB png spectrograms
- **batch_size=1** – pytorch parallelization parameter
- **num_workers=12** – pytorch parallelization parameter
- **apply_softmax=True** – if True, performs a softmax on raw output of network

returns: df of predictions indexed by file

7.16 Raven

raven.py: Utilities for dealing with Raven files

opensoundscape.raven.**annotation_check**(*directory*)

Check Raven annotations files for a non-null class

Input: directory: The path which contains Raven annotations file

Output: None

opensoundscape.raven.**generate_class_corrections**(*directory*)
Generate a CSV to specify any class overrides

Input: directory: The path which contains Raven annotations files ending in *.selections.txt.lower

Output:

csv (string): A multiline string containing a CSV file with two columns *raw* and *corrected*

opensoundscape.raven.**lowercase_annotations**(*directory*)
Convert Raven annotation files to lowercase

Input: directory: The path which contains Raven annotations file

Output: None

opensoundscape.raven.**query_annotations**(*directory*, *cls*)
Given a directory of Raven annotations, query for a specific class

Input: directory: The path which contains Raven annotations file cls: The class which you would like to query for

Output: output (string): A multiline string containing annotation file and rows matching the query cls

7.17 Species Table

7.18 Spectrogram

spectrogram.py: Utilities for dealing with spectrograms

class opensoundscape.spectrogram.**Spectrogram**(*spectrogram*, *frequencies*, *times*)
Immutable spectrogram container

amplitude(*freq_range=None*)
create an amplitude vs time signal from spectrogram

by summing pixels in the vertical dimension

Args freq_range=None: sum Spectrogrm only in this range of [low, high] frequencies in Hz (if None, all frequencies are summed)

Returns a time-series array of the vertical sum of spectrogram value

bandpass(*min_f*, *max_f*)
extract a frequency band from a spectrogram

crops the 2-d array of the spectrograms to the desired frequency range

Parameters

- **min_f** – low frequency in Hz for bandpass
- **high_f** – high frequency in Hz for bandpass

Returns bandpassed spectrogram object

classmethod from_audio(*audio*, *window_type='hann'*, *window_samples=512*, *overlap_samples=256*, *decibel_limits=(-100*, *-20)*)
create a Spectrogram object from an Audio object

Parameters

- **window_type="hann"** – see scipy.signal.spectrogram docs for description of window parameter
- **window_samples=512** – number of audio samples per spectrogram window (pixel)
- **overlap_samples=256** – number of samples shared by consecutive windows
- **=** (*decibel_limits*) – limit the dB values to (min,max) (lower values set to min, higher values set to max)

Returns opensoundscape.spectrogram.Spectrogram object

classmethod from_file()
create a Spectrogram object from a file

Parameters **file** – path of image to load

Returns opensoundscape.spectrogram.Spectrogram object

limit_db_range(*min_db=-100*, *max_db=-20*)
Limit the decibel values of the spectrogram to range from min_db to max_db

values less than min_db are set to min_db values greater than max_db are set to max_db

similar to Audacity's gain and range parameters

Parameters

- **min_db** – values lower than this are set to this
- **max_db** – values higher than this are set to this

Returns Spectrogram object with db range applied

linear_scale(*feature_range=(0, 1)*)
Linearly rescale spectrogram values to a range of values using in_range as decibel_limits

Parameters **feature_range** – tuple of (low,high) values for output

Returns Spectrogram object with values rescaled to feature_range

min_max_scale(*feature_range=(0, 1)*)
Linearly rescale spectrogram values to a range of values using in_range as minimum and maximum

Parameters **feature_range** – tuple of (low,high) values for output

Returns Spectrogram object with values rescaled to feature_range

net_amplitude(*signal_band*, *reject_bands=None*)
create amplitude signal in signal_band and subtract amplitude from reject_bands

rescale the signal and reject bands by dividing by their bandwidths in Hz (amplitude of each reject_band is divided by the total bandwidth of all reject_bands. amplitude of signal_band is divided by badwidth of signal_band.)

Parameters

- **signal_band** – [low,high] frequency range in Hz (positive contribution)
- **band** (*reject*) – list of [low,high] frequency ranges in Hz (negative contribution)

return: time-series array of net amplitude

plot(*inline=True*, *fname=None*, *show_colorbar=False*)
Plot the spectrogram with matplotlib.pyplot

Parameters

- **inline=True** –
- **fname=None** – specify a string path to save the plot to (ending in .png/.pdf)
- **show_colorbar** – include image legend colorbar from pyplot

to_image(*shape=None, mode='RGB', spec_range=[-100, -20]*)
create a Pillow Image from spectrogram linearly rescales values from db_range (default [-100, -20]) to [255,0] (ie, -20 db is loudest -> black, -100 db is quietest -> white)

Parameters

- **destination** – a file path (string)
- **shape=None** – tuple of image dimensions, eg (224,224)
- **mode="RGB"** – RGB for 3-channel color or "L" for 1-channel grayscale
- **spec_range=[-100,-20]** – the lowest and highest possible values in the spectrogram

Returns Pillow Image object

trim(*start_time*, *end_time*)
extract a time segment from a spectrogram

Parameters

- **start_time** – in seconds
- **end_time** – in seconds

Returns spectrogram object from extracted time segment

7.19 Taxa

a set of utilites for converting between scientific and common names of bird species in different naming systems (xeno canto and bird net)

opensoundscape.taxa.**bn_common_to_sci**(*common*)
convert bird net common name (ignoring dashes, spaces, case) to scientific name as lowercase-hyphenated

opensoundscape.taxa.**common_to_sci**(*common*)
convert bird net common name (ignoring dashes, spaces, case) to scientific name as lowercase-hyphenated

opensoundscape.taxa.**get_species_list**()
list of scientific-names (lowercase-hyphenated) of species in the loaded species table

opensoundscape.taxa.**sci_to_bn_common**(*scientific*)
convert scientific name as lowercase-hyphenated to birdnet common name as lowercasenospaces

opensoundscape.taxa.**sci_to_xc_common**(*scientific*)
convert scientific name as lowercase-hyphenated to xeno-canto common name as lowercasenospaces

opensoundscape.taxa.**xc_common_to_sci**(*common*)
convert xeno-canto common name (ignoring dashes, spaces, case) to scientific name as lowercase-hyphenated

7.20 Torch Spectrogram Augmentation

These functions were implemented for PyTorch in the following repository https://github.com/zcaceres/spec_augment
The original paper is available on https://arxiv.org/abs/1904.08779

7.21 Torch Training

opensoundscape.torch.train.**train**(*save_dir*, *model*, *train_dataset*, *valid_dataset*, *optimizer*, *loss_fn*, *epochs=25*, *batch_size=1*, *num_workers=0*, *log_every=5*, *tensor_augment=False*, *debug=False*, *print_logging=True*)

Train a model

Input: save_dir: A directory to save intermediate results model: A binary torch model,

- e.g. torchvision.models.resnet18(pretrained=True)
- must override classes, e.g. model.fc = torch.nn.Linear(model.fc.in_features, 2)

train_dataset: The training Dataset, e.g. created by SingleTargetAudioDataset() valid_dataset: The validation Dataset, e.g. created by SingleTargetAudioDataset() optimizer: A torch optimizer, e.g. torch.optim.SGD(model.parameters(), lr=1e-3) loss_fn: A torch loss function, e.g. torch.nn.CrossEntropyLoss() epochs: The number of epochs [default: 25] batch_size: The size of the batches [default: 1] num_workers: The number of cores to use for batch preparation [default: 1] log_every: Log statistics when epoch % log_every == 0 [default: 5] tensor_augment: Whether or not to use the tensor augment procedures [default: False] debug: Whether or not to write intermediate images [default: False]

Side Effects: Write a file *epoch-{epoch}.tar* containing (rate of *log_every*): - Model state dictionary - Optimizer state dictionary - Labels in YAML format - Train: loss, accuracy, precision, recall, and f1 score - Validation: accuracy, precision, recall, and f1 score - train_dataset.label_dict Write a metadata file with parameter values to save_dir/metadata.txt

Output: None

Effects: model parameters are saved to

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X