Comparing the effectiveness of CNN and logistic regression for increasingly difficult bird species classification

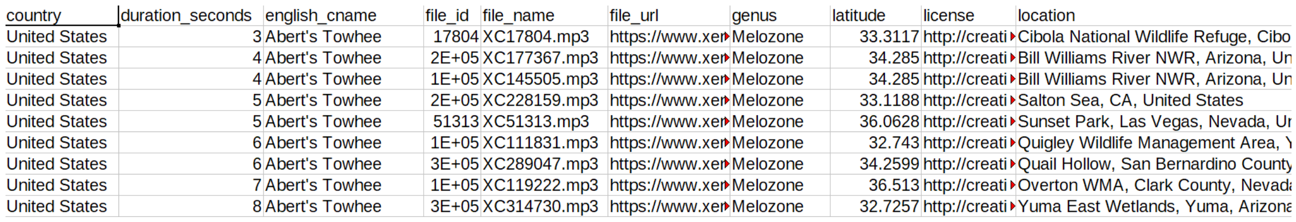
Introduction

From an ecological and environmental point of view, monitoring bird diversity is an important task. While bird monitoring is a well-established process, the observation is largely carried out manually which is time-consuming, and hence the scalability is low. This has motivated the use of machine learning methods to analyze bird images and sounds, using camera-trap data, recorded data or crowd-sourcing.

The aim of this report is to investigate the effectiveness of classifying bird species based on their songs. We begin by formalizing the problem, after which the applied machine learning methods are discussed. After that, results are analyzed along with the conclusion of findings. There is an appendix at the end containing the code used to produce the results.

Problem formulation

Given an unprocessed audio recording that has a bird present, the goal is to train a ML classifier that assigns it a category from a known list of species. The list of species to choose from will be gradually increased, and the change in model performance graphed. For this, the project uses a dataset of North American birds, created by the online community Xeno Canto [1]. The dataset is licensed under Creative Commons, and is available at Kaggle [2].



country	duration_seconds	english_cname	file_id	file_name	file_url	genus	latitude	license	location
United States	3	Abert's Towhee	17804	XC17804.mp3	https://www.xer	Melozone	33.3117	http://creati	Cibola National Wildlife Refuge, Cibo
United States	4	Abert's Towhee	2E+05	XC177367.mp3	https://www.xer	Melozone	34.285	http://creati	Bill Williams River NWR, Arizona, Un
United States	4	Abert's Towhee	1E+05	XC145505.mp3	https://www.xer	Melozone	34.285	http://creati	Bill Williams River NWR, Arizona, Un
United States	5	Abert's Towhee	2E+05	XC228159.mp3	https://www.xer	Melozone	33.1188	http://creati	Salton Sea, CA, United States
United States	5	Abert's Towhee	51313	XC51313.mp3	https://www.xer	Melozone	36.0628	http://creati	Sunset Park, Las Vegas, Nevada, Ur
United States	6	Abert's Towhee	1E+05	XC111831.mp3	https://www.xer	Melozone	32.743	http://creati	Quigley Wildlife Management Area, Y
United States	6	Abert's Towhee	3E+05	XC289047.mp3	https://www.xer	Melozone	34.2599	http://creati	Quail Hollow, San Bernardino County
United States	7	Abert's Towhee	1E+05	XC119222.mp3	https://www.xer	Melozone	36.513	http://creati	Overton WMA, Clark County, Nevada
United States	8	Abert's Towhee	3E+05	XC314730.mp3	https://www.xer	Melozone	32.7257	http://creati	Yuma East Wetlands, Yuma, Arizona

Figure 1. snapshot of the dataset

The set has 2730 datapoints, each containing the name of the species, an associated .mp3 file, the recording location etc. There are 91 species, equally distributed (30 datapoints per species), with recordings ranging from 1 – 190 seconds. For this project, the names of the species serve as labels, and features are extracted from a carefully chosen 2 second window of audio. As such, recordings shorter than 2 seconds have been discarded, giving a total size of 2714 datapoints.

Methods

As each recording could have a different length, different volume levels, a different sampling rate and varying amounts of silence, preprocessing was required. First, audio was resampled and compressed to a uniform scale of [-1, 1], 22050 samples/second. After this, a window selection algorithm was run to detect an active window from the recording, by searching for large absolute changes in volume, and finding their center.

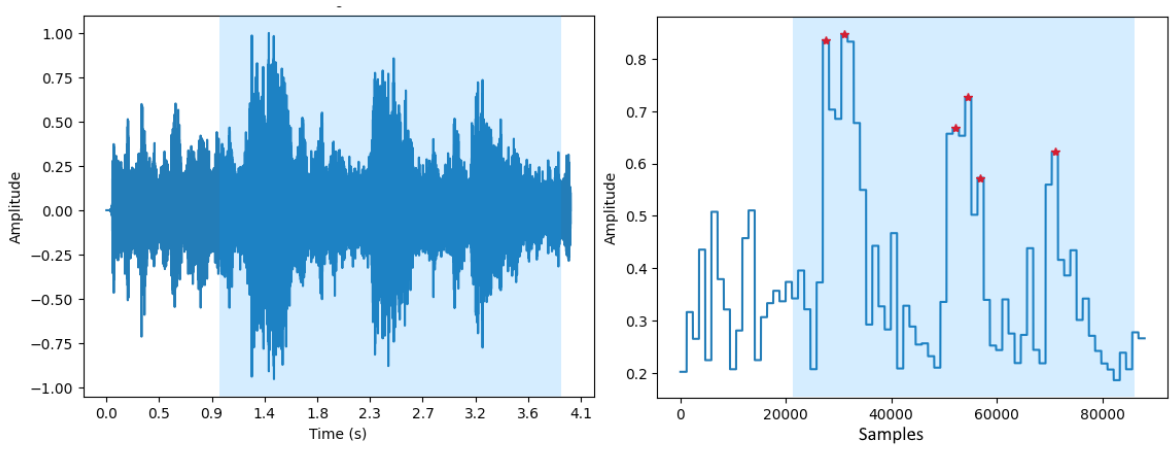


Figure 2. Window selection algorithm

After this, 7 different audio feature extraction methods [3] were considered and chosen for study.

1. Downsampled audio data (1 x 256)
2. Spectral centroid of the signal (1 x 128)
3. Spectral bandwidth of the signal (1 x 128)
4. Chromagram (128 x 128)
5. Mel spectogram (128 x 128)
6. Mel-frequency cepstral coefficients (128 x 128)
7. Short-time Fourier transform (128 x 128)

(The matrix dimensions of each feature is listed inside brackets)

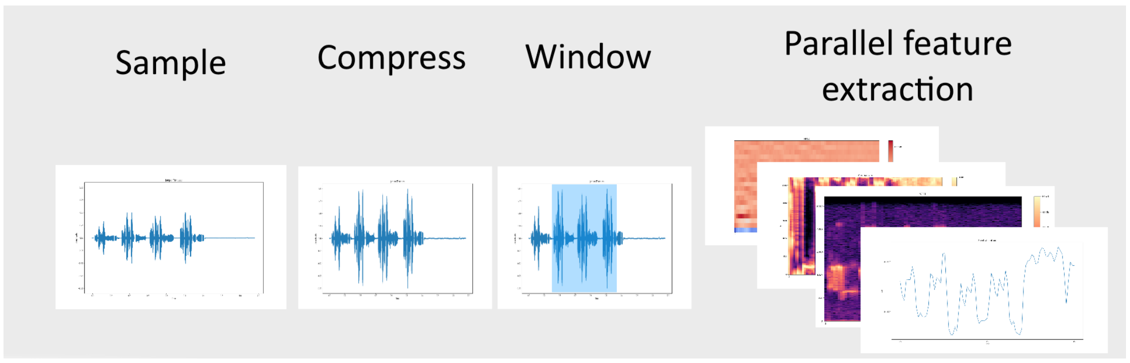


Figure 3. Preprocessing and feature extraction pipeline. Multiprocessing was used to speed up execution time.

For modeling the problem, logistic regression was chosen as a starting point. This follows a hypothesis, that there exists a linear map from some of the audio feature spaces to the categories of bird species. While this is unlikely to be true for the time domain signal, such a map could exist in the frequency domain.

To test logistic regression, random subsets of the original 2714 datapoints were created by filtering out species, effectively reducing the scope of the problem. After this, the accuracy of models trained for different features were graphed against the number of species they had to try classifying. The datapoints provided for training and validation were split 80-20%. The loss function used was logistic loss [4].

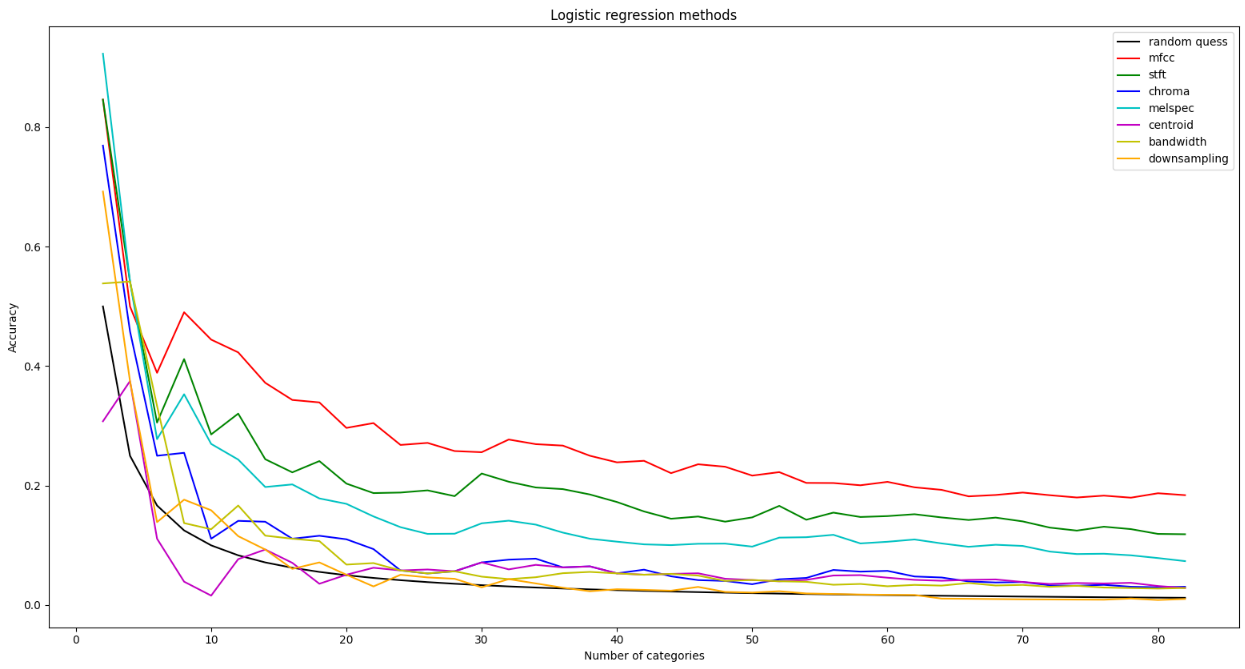


Figure 4. Logistic regression performance for different considered features.

The second tested model was convolutional neural networks, implemented with Tensorflow [5]. These were chosen to model a nonlinear hypothesis. The choice was motivated by similarity of complexity to image classification, as well as existing research applying them to audio classification [6].

As features, the CNN could take advantage of both matrix features (128 x 128, such as the mel spectogram) or combinations of matrix features (tensor consisting of chromagram, mfcc and stft). Different CNN architectures were tested similarly to logistic regression, using subsets of 4, 8, 16, 20 and 40 species. Each architecture was allowed to train for 60 epochs, first using categorical hinge loss, and then using categorical cross entropy.

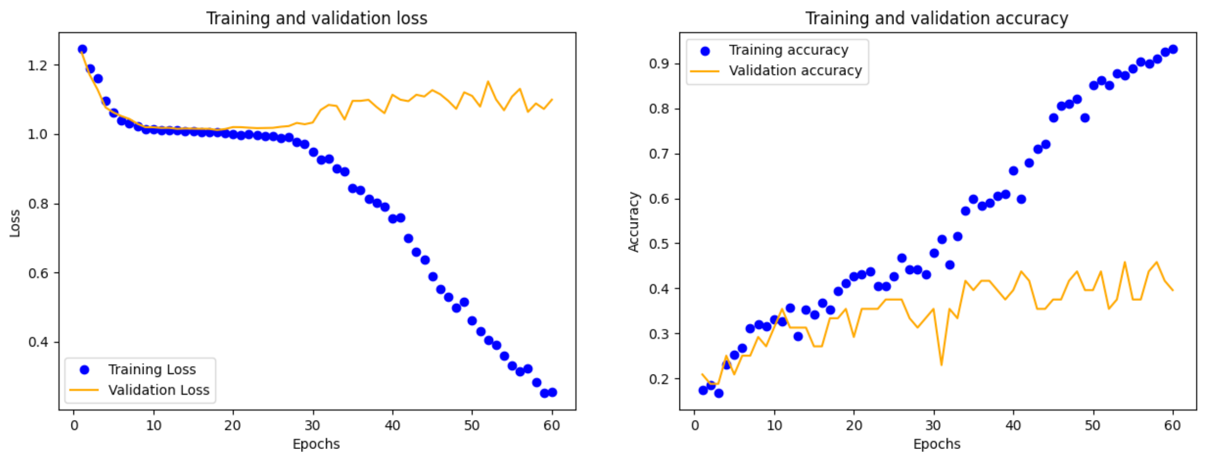


Figure 5. Example of a CNN model training

Results

The CNN methods suffered from overall poor performance due to overfitting, quickly achieving 100% accuracy on the test set, but around 33-44% for the validation. Out of 10 different tested architectures, shallower architectures tended to perform slightly better. The choice between hinge and crossentropy did not significantly affect accuracy, however hinge was slower to train. As for features, chromagram, stft and mel spectogram outclassed the other feature selection methods, while a combination of the three yielded, on average, 8% more accuracy. However, these findings may not be entirely trustworthy, as choosing different species for consideration affected the results of repeated tests. Most notably, models showed a large drop in performance at 20 species for an unknown reason.

Logistic regression performed reasonably well for classifying few species (2 - 8), and showed clear and repeatable results when either mfcc, stft or mel spectogram were used as features. It's performance after 10 species dropped to the 30-40% region.

When comparing logistic regression and CNN for few species classification (4 – 8) their performance was very similar. Using the best determined parameters for both models, CNN achieved marginally better results.

In conclusion, while the tested models showed some predictive power, there is a lot room for improvement. Models trained to only classify a few species gave more repeatable results, suggesting that an ensemble classifier could be implemented. Between CNN and logistic regression, logistic regression showed only slightly worse accuracy to CNN when using mel frequency cepstral coefficients, but was faster to train. As such, it is the current suggested method for the problem, with an average validation accuracy of 63% for 8 or less species.

References

[1] Willem-Pier Vellinga, ”Xeno-canto - Bird sounds from around the world”. Xeno-canto Foundation for Nature Sounds, 2022. doi: 10.15468/QV0KSN.

[2] ”Avian Vocalizations from CA & NV, USA”. https://www.kaggle.com/datasets/samhiatt/xenocanto-avian-vocalizations-canv-usa

[3] ”Feature extraction — librosa 0.10.0.dev0 documentation”. https://librosa.org/doc/main/feature.html

[4] ”Logistic Regression: Loss and Regularization | Machine Learning”, Google Developers. https://developers.google.com/machine-learning/crash-course/logistic-regression/model-training

[5] ”TensorFlow”, TensorFlow. https://www.tensorflow.org/

[6] S. Hershey ym., ”CNN Architectures for Large-Scale Audio Classification”, teoksessa International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2017. [Internet]. Available at: https://arxiv.org/abs/1609.09430

[7] P. Nandi, ”CNNs for Audio Classification”, Medium, 10. December 2021. https://towardsdatascience.com/cnns-for-audio-classification-6244954665ab

[8] ”Classify MNIST Audio using Spectrograms/Keras CNN”. https://kaggle.com/code/christianlillelund/classify-mnist-audio-using-spectrograms-keras-cnn

Appendix

main

October 9, 2022

[ ]:
```
import random
import numpy as np
import pandas as pd

import bird_utility
import bird_processing

import time
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from bird_learn import logistic_learn, CNN_learn
from bird_parallel_learn import parallel_learners


import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator

#--------------------Parameters---------------------#

default_seed = 42
training_split = 0.2
csv_path = 'xeno-canto/xeno-canto_ca-nv_index.csv'

# Discard audio files shorter than this
# Also used as window duration
min_duration = 2

# Supported feature extraction methods
feature_extraction_methods = [

    # Logistic regression compatible
    'mfcc',
    'stft',
    'chroma',
    'melspec',
    'centroid',
```

```python
    'bandwidth',
    'downsampling',

    # CNN only
    'spectral-tensor',
    'time-tensor'

]


#---------------------------------------------------#


#---------------------Methods----------------------#

# Prepare data from index. Choose the scope of the classification problem
# by only considering n species from the dataset
def data_preparation(num_species, seed=default_seed):

    n_species = num_species

    # Open index and filter columns we don't need
    csv = pd.read_csv(csv_path)
    csv = csv[['english_cname', 'file_name', 'duration_seconds']]

    # Discard audio files that are too short
    csv = csv[csv['duration_seconds'] >= min_duration]
    csv = csv.drop(columns=['duration_seconds'])

    csv.columns = ['name', 'file']
    print(f'Number of datapoints in index: {len(csv)}')

    # Optionally filter species to reduce scope
    if n_species > 0:
        species = csv['name'].drop_duplicates().values.tolist()
        random.Random(seed).shuffle(species)
        chosen_species = species[:n_species]
        csv = csv[csv['name'].isin(chosen_species)]
    else:
        # Set the value to the amount of unique species
        n_species = len(csv['name'].drop_duplicates().values.tolist())

    # Transform labels to numerical categories
    label_encoder = LabelEncoder().fit(csv['name'])
    labels = label_encoder.transform(csv['name'].to_numpy())

    # Binarizer for one hot encoding
    label_binarizer = LabelBinarizer().fit(labels)

    # Split dataset
    X_train_f, X_val_f, y_train, y_val = train_test_split(csv['file'].
 ↪to_numpy(), labels, test_size=training_split, random_state=seed)
    print(f'Set sizes train/validation: [{len(X_train_f)}, {len(X_val_f)}]')
    return X_train_f, X_val_f, y_train, y_val, label_encoder, label_binarizer


# Given prepared datapoints, create a subset where categories have been removed
# This is to reduce redundant feature extraction work.
def subset_datapoints(X_train, X_val, y_train, y_val, n_species):
    classes = np.unique(np.concatenate((y_train, y_val)))
    if n_species >= len(classes):
        return X_train, X_val, y_train, y_val
    else:
        chosen_species = classes[:n_species]
        X_train_n = np.array([X_train[i] for i in range(len(y_train)) if␣
 ↪(y_train[i] in chosen_species)])
        X_val_n = np.array([X_val[i] for i in range(len(y_val)) if (y_val[i] in␣
 ↪chosen_species)])
        y_train_n = np.array([v for v in y_train if (v in chosen_species)])
        y_val_n = np.array([v for v in y_val if (v in chosen_species)])
        print(f'Set sizes train/validation: [{len(X_train_n)}, {len(X_val_n)}]')
        return X_train_n, X_val_n, y_train_n, y_val_n


# Test feature extration and confirm the shapes of processed data
def feature_extraction_showcase(file):
    windowed = bird_processing.preprocessing(file)
    sr = 22050
    bird_utility.audio_plot(windowed, sr, 'Original')
    bird_utility.audio_plot(bird_processing.audio_downsample(windowed, 256),␣
 ↪256 // 2, 'Downsampled')
    bird_utility.spectral_power_plot(bird_processing.
 ↪audio_melspectrogram(windowed, sr), sr, 'Melspectrogram')
    bird_utility.mfcc_plot(bird_processing.audio_mfcc(windowed, sr), sr, 'MFCC')
    bird_utility.spectral_amplitude_plot(bird_processing.audio_stft(windowed),␣
 ↪sr, 'STFT')
    bird_utility.spectral_amplitude_plot(bird_processing.audio_chroma(windowed,␣
 ↪sr), sr, 'Chromagram')
    bird_utility.centroid_plot(bird_processing.audio_centroid(windowed, sr),␣
 ↪sr, 'Spectral centroid')
    bird_utility.centroid_plot(bird_processing.audio_spectral_band(windowed,␣
 ↪sr), sr, 'Spectral bandwidth')
```

```
    # Confirm shapes
    print('\nFeature shapes')
    print(f'original shape: {windowed.shape}')
    print(f'downsampled shape: {bird_processing.audio_downsample(windowed, 256).
  ↪shape}')
    print(f'melspec shape: {bird_processing.audio_melspectrogram(windowed, sr).
  ↪shape}')
    print(f'mfcc shape: {bird_processing.audio_mfcc(windowed, sr).shape}')
    print(f'stft shape: {bird_processing.audio_stft(windowed).shape}')
    print(f'chroma shape: {bird_processing.audio_chroma(windowed, sr).shape}')
    print(f'centroid shape: {bird_processing.audio_centroid(windowed, sr).
  ↪shape}')
    print(f'spec band shape: {bird_processing.audio_spectral_band(windowed, sr).
  ↪shape}')


# Given datapoints from the index, generate a feature matrix using some method
def generate_feature_matrix(X_train_f, X_val_f, chosen_method='mfcc'):
    print(f"Extracting feature vectors: method='{chosen_method}'")

    t_start = time.time()
    X_train = bird_processing.extract_features(X_train_f, chosen_method)
    print(f'Time elapsed: {"%.2f" % (time.time() - t_start)} s')

    t_start = time.time()
    X_val = bird_processing.extract_features(X_val_f, chosen_method)
    print(f'Time elapsed: {"%.2f" % (time.time() - t_start)} s')

    print('Feature extraction complete')
    print(f'Received flattened feature data: [X_train={X_train.shape},␣
  ↪X_val={X_val.shape}]')
    return X_train, X_val


# Train logistic regressors for multiple methods and compare the results
# Graph model performace compared to problem scope (n_species)
def logistic_training_montage(max_species, min_species=2, step=1, n_methods=2,␣
  ↪seed=default_seed):

    X_train_files, X_val_files, y_train, y_val, label_encoder, label_binarizer␣
  ↪= data_preparation(num_species=max_species, seed=seed)
    N = (max_species - min_species) // step + 1
    results = np.zeros(shape=(n_methods, N))
    colors = ['r', 'g', 'b', 'c', 'm', 'y', 'orange', 'purple']

    # Test all methods
    for m in range(n_methods):
        X_train, X_val = generate_feature_matrix(X_train_files, X_val_files,␣
↪feature_extraction_methods[m])

        # Learn logistic regressors in parallel
        t_start = time.time()
        metrics_vector = parallel_learners(X_train, X_val, y_train, y_val,␣
↪max_species, min_species, step)
        print(f'Time elapsed: {"%.2f" % (time.time() - t_start)} s')

        # Add results
        for s in range(N):
            results[m][s] = metrics_vector[s][0] # accuracy

        # Write metrics to file
        with open(f'metrics-{feature_extraction_methods[m]}.txt', mode = 'w')␣
↪as out:
            out.write(f'Testing {feature_extraction_methods[m]} for categories␣
↪[{min_species} => {max_species}]\n')
            out.write(f'Step size {step}\n')
            for metric in metrics_vector:
                for v in metric:
                    out.write(f'{v} ')
                out.write('\n')

    fig, ax = plt.subplots()
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    ax.set_title('Logistic regression methods')
    plt.xlabel('Number of categories')
    plt.ylabel('Accuracy')

    guess = [1.0 / i for i in range(min_species, max_species + 1, step)]
    ax.plot(range(min_species, max_species + 1, step), guess, 'k',␣
↪label='random guess')

    for m in range(n_methods):
        ax.plot(range(min_species, max_species + 1, step), results[m],␣
↪colors[m], label=feature_extraction_methods[m])
    ax.legend()
    plt.savefig('multi-comp.pdf')
    plt.show()



# Compare logistic regression and CNN
def logistic_CNN_comparison(max_species, min_species=2, step=1,
                              logistic_method='mfcc',␣
↪CNN_method='spectral-tensor',
                              CNN_architecture='CNN_S128', CNN_input_shape=(128,␣
↪128, 3), seed=default_seed):

    X_train_files, X_val_files, y_train, y_val, label_encoder, label_binarizer␣
↪= data_preparation(num_species=max_species, seed=seed)
    N = (max_species - min_species) // step + 1
    results = np.zeros(shape=(2, N))
    colors = ['r', 'b']

    X_train_l, X_val_l = generate_feature_matrix(X_train_files, X_val_files,␣
↪logistic_method)
    X_train_cnn, X_val_cnn = generate_feature_matrix(X_train_files,␣
↪X_val_files, CNN_method)

    # Learn logistic regressors in parallel
    t_start = time.time()
    metrics_vector = parallel_learners(X_train_l, X_val_l, y_train, y_val,␣
↪max_species, min_species, step)
    print(f'Time elapsed: {"%.2f" % (time.time() - t_start)} s')

    # Add results
    for s in range(N):
        results[0][s] = metrics_vector[s][0] # accuracy

    # Write metrics to file
    with open(f'metrics-{logistic_method}.txt', mode = 'w') as out:
        out.write(f'Testing {logistic_method} for categories [{min_species} =>␣
↪{max_species}]\n')
        out.write(f'Step size {step}\n')
        for metric in metrics_vector:
            for v in metric:
                out.write(f'{v} ')
            out.write('\n')

    # Learn CNNs (one at a time, but gpu boosted)
    for num_spec in range(min_species, max_species, step):

        X_train_c, X_val_c, y_train_c, y_val_c = subset_datapoints(X_train_cnn,␣
↪X_val_cnn, y_train, y_val, num_spec)

        # Actually need to refit the binarizer so the y shape matches the␣
↪generated CNN architecture
        print(f'Shape concat {np.concatenate((y_train_c, y_val_c)).shape}')
        print(f'num unique {len(np.unique(np.concatenate((y_train_c,␣
↪y_val_c))))}')

        binarizer = LabelBinarizer().fit(np.concatenate((y_train_c, y_val_c)))

        X_train_c = np.array([np.reshape(feature, CNN_input_shape) for feature␣
↪in X_train_c])
        X_val_c = np.array([np.reshape(feature, CNN_input_shape) for feature in␣
↪X_val_c])

        y_train_c = binarizer.transform(y_train_c)
        y_val_c = binarizer.transform(y_val_c)
        print(y_train_c.shape)
        print(y_val_c.shape)

        metrics = CNN_learn(X_train_c, y_train_c, X_val_c, y_val_c,␣
↪CNN_input_shape, num_spec, 50, CNN_architecture, plots=False)
        results[1][(num_spec - min_species) // step] = metrics['accuracy']

    fig, ax = plt.subplots()
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    ax.set_title('Logistic regression vs CNN')
    plt.xlabel('Number of categories')
    plt.ylabel('Accuracy')

    guess = [1.0 / i for i in range(min_species, max_species + 1, step)]
    ax.plot(range(min_species, max_species + 1, step), guess, 'k',␣
↪label='random guess')

    ax.plot(range(min_species, max_species + 1, step), results[0], colors[0],␣
↪label=logistic_method)
    ax.plot(range(min_species, max_species + 1, step), results[1], colors[1],␣
↪label=CNN_method)
    ax.legend()
    plt.savefig('log-cnn-comp.pdf')
    plt.show()




#----------------Execution starts here--------------#

#feature_extraction_showcase('XC469889.mp3')

# Logistic regression
"""
```

```
X_train_f, X_val_f, y_train, y_val, label_encoder, label_binarizer =␣
 ↪data_preparation(num_species=10)
X_train, X_val = generate_feature_matrix(X_train_f, X_val_f, 'mfcc')

logistic_learn(X_train, y_train, X_val, y_val)
"""

# Warning, reduce the parameters unless you want to leave running overnight
#logistic_training_montage(max_species=82, min_species=2, step=2, n_methods=7,␣
 ↪seed=933)


#logistic_CNN_comparison(20, 3, 1, CNN_architecture='CNN_D128', seed=853)


# CNN testing

num_spec = 8

X_train_f, X_val_f, y_train_c, y_val_c, label_encoder, label_binarizer =␣
 ↪data_preparation(num_species=num_spec, seed=92)
X_train_c, X_val_c = generate_feature_matrix(X_train_f, X_val_f, 'stft')

X_train_c = np.array([feature.reshape(128, 128, 1) for feature in X_train_c])
X_val_c = np.array([feature.reshape(128, 128, 1) for feature in X_val_c])

y_train_c = label_binarizer.transform(y_train_c)
y_val_c = label_binarizer.transform(y_val_c)

CNN_learn(X_train_c, y_train_c, X_val_c, y_val_c, (128, 128, 1), num_spec, 60,␣
 ↪'CNN_D128')
```

bird_processing

October 9, 2022

[ ]:
```python
# Multicore boosted signal processing
import os
import sys
import numpy as np
import librosa as lr
import audioread.ffdec
import multiprocessing as mp
from subprocess import Popen, PIPE
from scipy.signal import find_peaks


#--------------------Parameters---------------------#

# Change this to reduce cpu load
cores = mp.cpu_count()

# Resample audio to a common rate when opening
resampling = 22050

# Constant for peak finding algorithm
binary_filter_window = 150

# Window size to cut in seconds
window_size = 2

# Path to audio files
audio_path = 'xeno-canto/xeno-canto-ca-nv/'

# FFT window length
fft_window = 1024

# Targets for fitting the audio features to the shape of the 2D CNN
# Don't change these
component_target = 128
sample_target = 128

# Calculate hop_length needed to fit targets
```

```
hop_length_target = resampling * window_size // sample_target + 1

# Normalize outputs
# For each feature, center at 0 and divide by std deviation
normalize_features = True

#--------------------------------------------------#





#----------------Signal processing----------------#


# IMPORTANT requires access to ffmpeg in order to work
# Either add it to PATH or provide it in the root folder
def read_mp3(filepath):
    f = audioread.ffdec.FFmpegAudioFile(audio_path+filepath)
    return lr.load(f, sr=resampling if resampling > 0 else f.samplerate)


# Normalize audio to [-1, 1]
def audio_normalize(data):
    return lr.util.normalize(data, axis=0, fill=True)


# Downsample audio to N samples
def audio_downsample(data, samples):
    d = int(np.floor(len(data)/samples))
    vals = np.arange(0, d*samples, d)
    return np.array([data[i:(i+d)][0] for i in vals])



# Quantize audio to remove noise for peak finding
def audio_binary_filter(data, samples):
    d = int(np.floor(len(data)/samples))
    vals = np.arange(0, d*samples, samples)
    filtered = np.array([np.max(data[i:(i+samples)])*np.ones((samples)) for i␣
 ↪in vals])
    return filtered.flatten()



# Given a signal, identify the central area via analyzing peak amplitudes
def peak_finder(data):
    # Filter the signal before peak finding
    peaks = []
    sampling = 2*len(data) // binary_filter_window
    reduced = audio_binary_filter(data, sampling)

    # Initial treshold to consider peaks
    treshold = np.mean(reduced) + 1*np.std(reduced)

    # Find a minimum of 3 peaks by continually reducing the threshold
    while len(peaks) < 3:
        peaks, properties = find_peaks(reduced, height=(treshold, np.
 ↪max(reduced)))
        treshold -= 10

    return peaks[len(peaks) // 2]



# Select an appropriate window from the recording
def audio_window(data, sampling_rate):

    window = window_size * sampling_rate

    # pad with mean value if we are too short
    # although this should never be the case
    if len(data) < window:
        mean = np.mean(data)
        return np.append(data, mean*np.ones((window-len(data))))
    elif len(data) == window:
        return data
    else:
        window_center = peak_finder(data)
        start = window_center - window // 2
        if start <= 0: return data[:window]

        end = window_center + window // 2
        if end >= len(data): return data[-window:]

        return data[start:end]




# Mel Frequency Cepstral Coefficients

def audio_mfcc(data, sampling_rate):
    return lr.feature.mfcc(y=data, sr=sampling_rate, n_fft=fft_window,␣
 ↪n_mfcc=component_target, hop_length=hop_length_target)

def audio_melspectrogram(data, sampling_rate):
    return lr.feature.melspectrogram(y=data, sr=sampling_rate,␣
 ↪n_fft=fft_window, n_mels=component_target, hop_length=hop_length_target)


# Short Time Fourier Transform
def audio_stft(data):
    return np.abs(lr.stft(y=data, n_fft=fft_window // 4 - 1,␣
 ↪hop_length=hop_length_target))

# Chromagram
def audio_chroma(data, sampling_rate):
    return lr.feature.chroma_stft(y=data, sr=sampling_rate, n_fft=fft_window*4,␣
 ↪n_chroma=component_target, hop_length=hop_length_target)

# Spectral centroid
def audio_centroid(data, sampling_rate):
    return lr.feature.spectral_centroid(y=data, sr=sampling_rate,␣
 ↪hop_length=hop_length_target)

# Spectral bandwidth
def audio_spectral_band(data, sampling_rate):
    return lr.feature.spectral_bandwidth(y=data, sr=sampling_rate,␣
 ↪hop_length=hop_length_target)

#---------------------------------------------------#




#----------------Feature extraction-----------------#

# Preprocessing before feature extraction
def preprocessing(filename):

    data, sr = read_mp3(filename)

    # Compress sound and select a constant size window
    norm_data = lr.util.normalize(data, axis=0, fill=True)
    return audio_window(norm_data, sr)

# Normalize output to
def normalize(data):
```

```python
    return (data - data.mean()) / (data.std() + 1e-8)


# Generate features by downsampling original audio vector
# Downsample to 256 => 16x16
def feature_downsample(filename):
    data = audio_downsample(preprocessing(filename), 256)
    return normalize(data) if normalize_features else data

# Generate features by taking different audio metrics

def feature_mfcc(filename):
    data = audio_mfcc(preprocessing(filename), resampling)
    return normalize(data) if normalize_features else data

def feature_chroma(filename):
    data = audio_chroma(preprocessing(filename), resampling)
    return normalize(data) if normalize_features else data

def feature_melspec(filename):
    data = audio_melspectrogram(preprocessing(filename), resampling)
    return normalize(data) if normalize_features else data

def feature_stft(filename):
    data = audio_stft(preprocessing(filename))
    return normalize(data) if normalize_features else data

def feature_centroid(filename):
    data = audio_centroid(preprocessing(filename), resampling)
    return normalize(data) if normalize_features else data

def feature_bandwidth(filename):
    data = audio_spectral_band(preprocessing(filename), resampling)
    return normalize(data) if normalize_features else data

# Combine spectral components to form feature tensors of shape (N x N x 3)
def feature_spectral_tensor(filename):
    windowed = preprocessing(filename)
    c1 = audio_stft(windowed)
    c2 = audio_melspectrogram(windowed, resampling)
    c3 = audio_chroma(windowed, resampling)

    c1 = normalize(c1) if normalize_features else c1
    c2 = normalize(c2) if normalize_features else c2
    c3 = normalize(c3) if normalize_features else c3

    tensor = np.empty(shape=(component_target, sample_target, 3))
    for i in range(component_target):
        for j in range(sample_target):
            tensor[i][j][0] = c1[i][j]
            tensor[i][j][1] = c2[i][j]
            tensor[i][j][2] = c3[i][j]
    return tensor

# Combine time domain components to form tensors of shape (16, 8, 3)
def feature_time_tensor(filename):
    windowed = preprocessing(filename)
    c1 = audio_downsample(windowed, 128).reshape(16, 8)
    c2 = audio_centroid(windowed, resampling).reshape(16, 8)
    c3 = audio_spectral_band(windowed, resampling).reshape(16, 8)

    c1 = normalize(c1) if normalize_features else c1
    c2 = normalize(c2) if normalize_features else c2
    c3 = normalize(c3) if normalize_features else c3

    tensor = np.empty(shape=(16, 8, 3))
    for i in range(16):
        for j in range(8):
            tensor[i][j][0] = c1[i][j]
            tensor[i][j][1] = c2[i][j]
            tensor[i][j][2] = c3[i][j]
    return tensor


#----------------------------------------------------#





#-----------------Multiprocessing-------------------#

# Perform feature extraction accelerated by multiprocessing
# Call this function to spawn a child process that further
# distributes work among cpu cores
def extract_features(X_filenames, method):
    with Popen('python bird_processing.py', cwd=os.getcwd(), stdin=PIPE,␣
 ↪stdout=PIPE, universal_newlines=True) as p:
        # Send data
        with p.stdin as pipe:
            pipe.write(f'{method}\n')
            for filename in X_filenames:
                pipe.write(f'{filename}\n')
        data = []

        # Receive progress updates
        for line in p.stdout:
            l = line.strip()
            print(l)
            if l == 'Done':
                break

        # Receive data
        for line in p.stdout:
            data.append(line.strip())

        # Reformat
        data_vectors = np.array([[float(v) for v in d.split(' ')] for d in␣
 ↪data])
        return data_vectors



# Worker process main function

if __name__ == '__main__':

    input = []
    output = []
    for line in sys.stdin:
        input.append(line.strip())

    method = input[0]
    input.pop(0)

    # Choose feature generation method
    generate_features = (
        {
            'mfcc': feature_mfcc,
            'stft': feature_stft,
            'chroma': feature_chroma,
            'melspec': feature_melspec,
            'centroid': feature_centroid,
            'bandwidth': feature_bandwidth,
            'downsampling': feature_downsample,
            'spectral-tensor': feature_spectral_tensor,
            'time-tensor': feature_time_tensor
        }[method]
    )
    print('Processing...')
    sys.stdout.flush()
    with mp.Pool(cores) as p:
        output = p.map(generate_features, input)
    print('Done')


    # Send back data:
    for feature in output:
        vec = feature.flatten()
        for v in vec:
            print(v, end=' ')
        print()
        sys.stdout.flush()
```

bird_learn

October 9, 2022

```
[ ]: import numpy as np
     from keras.models import Sequential
     from keras.layers import Conv1D, MaxPooling1D, Conv2D, MaxPooling2D, Flatten,␣
      ↪Dense, LayerNormalization, Dropout
     from keras.losses import CategoricalCrossentropy, CategoricalHinge

     import matplotlib.pyplot as plt
     from sklearn.linear_model import LogisticRegression
     from sklearn.metrics import accuracy_score, precision_score, recall_score


     # If a method is worse than this,
     # then we have definitely failed
     def random_guess_accuracy(y_train, y_val):
         n_classes = len(np.unique(np.concatenate((y_train, y_val))))
         return 1.0 / n_classes



     # Different CNN architectures

     # Shallow 32
     def CNN_S32(input_shape, n_classes):
         model = Sequential()
         model.add(Conv2D(32, (3, 3), activation='relu', padding='same',␣
      ↪input_shape=input_shape))
         model.add(LayerNormalization())
         model.add(MaxPooling2D((2, 2), padding='same'))
         model.add(LayerNormalization())
         model.add(Flatten())
         model.add(Dense(32, activation='relu'))
         model.add(LayerNormalization())
         model.add(Dropout(0.2))
         model.add(Dense(n_classes, activation='softmax'))
         model.summary()
         return model
```

```python
# Shallow 64
def CNN_S64(input_shape, n_classes):
    model = Sequential()
    model.add(Conv2D(64, (3, 3), activation='relu', padding='same',␣
 ↪input_shape=input_shape))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())
    model.add(Flatten())
    model.add(Dense(64, activation='relu'))
    model.add(LayerNormalization())
    model.add(Dropout(0.2))
    model.add(Dense(n_classes, activation='softmax'))
    model.summary()
    return model

# Shallow 128
def CNN_S128(input_shape, n_classes):
    model = Sequential()
    model.add(Conv2D(128, (3, 3), activation='relu', padding='same',␣
 ↪input_shape=input_shape))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(LayerNormalization())
    model.add(Dropout(0.2))
    model.add(Dense(n_classes, activation='softmax'))
    model.summary()
    return model

# Deep triple convolution 32
def CNN_D32(input_shape, n_classes):
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', padding='same',␣
 ↪input_shape=input_shape))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())
    model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())
    model.add(Dropout(0.2))
    model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())
    model.add(Flatten())
    model.add(Dense(32, activation='relu'))
    model.add(LayerNormalization())
    model.add(Dropout(0.2))
    model.add(Dense(n_classes, activation='softmax'))
    model.summary()
    return model

# Deep triple convolution 64
def CNN_D64(input_shape, n_classes):
    model = Sequential()
    model.add(Conv2D(64, (3, 3), activation='relu', padding='same',␣
 ↪input_shape=input_shape))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())
    model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())
    model.add(Dropout(0.2))
    model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())
    model.add(Flatten())
    model.add(Dense(64, activation='relu'))
    model.add(LayerNormalization())
    model.add(Dropout(0.2))
    model.add(Dense(n_classes, activation='softmax'))
    model.summary()
    return model

# Deep triple convolution 128
def CNN_D128(input_shape, n_classes):
    model = Sequential()
    model.add(Conv2D(128, (3, 3), activation='relu', padding='same',␣
 ↪input_shape=input_shape))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())
    model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())
    model.add(Dropout(0.2))
    model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(LayerNormalization())
    model.add(Dropout(0.2))
    model.add(Dense(n_classes, activation='softmax'))
    model.summary()
    return model

# 128 => 32 convolvers, 7 => 3 kernel
def CNN_F128(input_shape, n_classes):
    model = Sequential()
    model.add(Conv2D(128, (7, 7), activation='relu', padding='same',␣
 ↪input_shape=input_shape))
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())
    model.add(Conv2D(64, (5, 5), activation='relu', padding='same'))
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())
    model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())
    model.add(Flatten())
    model.add(Dense(64, activation='relu'))
    model.add(Dense(n_classes, activation='softmax'))
    model.summary()
    return model

# Deep 32, 5 convolutions
def CNN_T32(input_shape, n_classes):
    model = Sequential()

    model.add(Conv2D(32, (3, 3), activation='relu', padding='same',␣
 ↪input_shape=input_shape))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())

    model.add(Dropout(0.2))

    model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())

    model.add(Dropout(0.5))

    model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())

    model.add(Dropout(0.2))

    model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())

    model.add(Dropout(0.2))

    model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())

    model.add(Flatten())
    model.add(Dense(32, activation='relu'))
    model.add(LayerNormalization())
    model.add(Dense(n_classes, activation='softmax'))
    model.summary()
    return model

# Deep 128, 5 convolutions
def CNN_T128(input_shape, n_classes):
    model = Sequential()

    model.add(Conv2D(128, (3, 3), activation='relu', padding='same',␣
 ↪input_shape=input_shape))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())

    model.add(Dropout(0.2))

    model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())
    model.add(Dropout(0.5))

    model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())

    model.add(Dropout(0.2))

    model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())

    model.add(Dropout(0.2))

    model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())

    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(LayerNormalization())
    model.add(Dense(n_classes, activation='softmax'))
    model.summary()
    return model

# Deep 128, 5x5 kernel, no dropout
def CNN_E128(input_shape, n_classes):
    model = Sequential()
    model.add(Conv2D(128, (5, 5), activation='relu', padding='same',␣
 ↪input_shape=input_shape))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())
    model.add(Conv2D(128, (5, 5), activation='relu', padding='same'))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())
    model.add(Conv2D(128, (5, 5), activation='relu', padding='same'))
    model.add(LayerNormalization())
    model.add(MaxPooling2D((2, 2), padding='same'))
    model.add(LayerNormalization())
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(LayerNormalization())
    model.add(Dense(n_classes, activation='softmax'))
    model.summary()
    return model

# 1D Convolution
def CNN_E1024(input_shape, n_classes):
    model = Sequential()
    model.add(Conv1D(1024, 3, activation='relu', padding='same',␣
 ↪input_shape=input_shape))
    model.add(LayerNormalization())
    model.add(MaxPooling1D(2, padding='same'))
    model.add(LayerNormalization())
    model.add(Conv1D(1024, 3, activation='relu', padding='same'))
    model.add(LayerNormalization())
    model.add(MaxPooling1D(2, padding='same'))
    model.add(LayerNormalization())
    model.add(Dropout(0.2))
    model.add(Conv1D(1024, 3, activation='relu', padding='same'))
    model.add(LayerNormalization())
    model.add(MaxPooling1D(2, padding='same'))
    model.add(LayerNormalization())
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(LayerNormalization())
    model.add(Dropout(0.2))
    model.add(Dense(n_classes, activation='softmax'))
    model.summary()
    return model

def CNN_learn(X_train, y_train, X_val, y_val, shape, n_classes, n_epochs=20,␣
 ↪model_name='CNN_S32', plots=True, name=''):
    fname = f'{model_name}_{n_classes}_{n_epochs}' if not name else name

    architectures = {
        'CNN_S32': CNN_S32,
        'CNN_S64': CNN_S64,
        'CNN_S128': CNN_S128,
        'CNN_D32': CNN_D32,
        'CNN_D64': CNN_D64,
        'CNN_D128': CNN_D128,
        'CNN_F128': CNN_F128,
        'CNN_T32': CNN_T32,
        'CNN_T128': CNN_T128,
        'CNN_E128': CNN_E128,
        'CNN_E1024': CNN_E1024
    }
    # Loss functions CategoricalHinge(),␣
↪CategoricalCrossentropy(from_logits=False)

    model = architectures[model_name](shape, n_classes)
    model.compile(optimizer='adam',␣
↪loss=CategoricalCrossentropy(from_logits=False), metrics=['accuracy'])
    training = model.fit(X_train, y_train, epochs=n_epochs,␣
↪validation_data=(X_val, y_val))


    loss_values = training.history['loss']
    acc_values = training.history['accuracy']

    val_loss_values = training.history['val_loss']
    val_acc_values = training.history['val_accuracy']

    best_acc = 0.0
    best_epoch = 0
    for i in range(len(acc_values)):
        a = acc_values[i] * val_acc_values[i]
        if a > best_acc:
            best_acc = a
            best_epoch = i + 1

    accuracy = val_acc_values[best_epoch - 1]
    rand_acc = 1.0 / n_classes


    # Write metrics to file
    with open(f'metrics-{fname}.txt', mode = 'w') as out:
        out.write(f'Best achieved accuracy at epoch={best_epoch}:\n')
        out.write(f'{accuracy} {rand_acc} {accuracy - rand_acc}\n')
        out.write(f'Overview of training:\n')
        for i in range(len(acc_values)):
            out.write(f'{loss_values[i]} ')
            out.write(f'{acc_values[i]} ')
            out.write(f'{val_loss_values[i]} ')
            out.write(f'{val_acc_values[i]} ')
            out.write('\n')


    if plots:
        #val_loss_avg = np.convolve(val_loss_values, np.ones(25)/25,␣
↪mode='same')
        #val_acc_avg = np.convolve(val_acc_values, np.ones(25)/25, mode='same')
        epochs = range(1,n_epochs+1)
        fig, (ax1, ax2) = plt.subplots(1,2,figsize=(15,5))
        ax1.plot(epochs,loss_values,'bo',label='Training Loss')
        ax1.plot(epochs,val_loss_values,'orange', label='Validation Loss')
        #ax1.plot(epochs,val_loss_avg,'red', label='Validation loss average')
        ax1.set_title('Training and validation loss')
        ax1.set_xlabel('Epochs')
        ax1.set_ylabel('Loss')
        ax1.legend()
        ax2.plot(epochs,acc_values,'bo', label='Training accuracy')
        ax2.plot(epochs,val_acc_values,'orange',label='Validation accuracy')
        #ax2.plot(epochs,val_acc_avg,'red',label='Validation accuracy average')
        ax2.set_title('Training and validation accuracy')
        ax2.set_xlabel('Epochs')
        ax2.set_ylabel('Accuracy')
        ax2.legend()
        plt.savefig(f'{fname}.pdf')
        plt.show()

    return {
        'accuracy': accuracy,
        'rand_accuracy': rand_acc,
        'improvement': (accuracy-rand_acc)
    }



def logistic_learn(X_train, y_train, X_val, y_val, prints=True):

    if prints: print(f'Training logistic regressor for X[{X_train.shape}] =>␣
 ↪y[{y_train.shape}]')

    # SAG solver suffered from convergence issues for some features
    #
    model = LogisticRegression(solver='liblinear', max_iter=1000)
    model.fit(X_train, y_train)
    if prints: print(f'Validating on set X[{X_val.shape}] => y[{y_val.shape}]')
    y_pred = model.predict(X_val)

    accuracy = accuracy_score(y_val, y_pred)
    rand_acc = random_guess_accuracy(y_train, y_val)

    # Use micro for multiclass scoring
    precision = precision_score(y_val, y_pred, average='micro')
    recall = recall_score(y_val, y_pred, average='micro')

    # Formula taken from scikit-learn.org
    f1 = 2 * (precision * recall) / (precision + recall)
    if prints:
        print('--- Model statistics ---')
        print()
        print(f'Model accuracy:  {"%.3f" % (100.0*accuracy)}%')
        print(f'== Random guess: {"%.3f" % (100.0*rand_acc)}%')
        print(f'== Improvement:  {"%.3f" % (100.0*(accuracy-rand_acc))}%')
        print()
        print(f'Precision:       {"%.3f" % precision}')
        print(f'Recall:          {"%.3f" % recall}')
        print(f'F1 Score:        {"%.3f" % f1}')
        print('------------------------\n')

    # Return metrics
    return {
        'accuracy': accuracy,
        'rand_accuracy': rand_acc,
        'improvement': (accuracy-rand_acc),
        'precision': precision,
        'recall': recall,
        'f1': f1
    }
```

bird_parallel_learn

October 9, 2022

[ ]:
```python
# Copy pasted parallel worker from bird_processing.py
import os
import sys
import numpy as np
import multiprocessing as mp
from subprocess import Popen, PIPE
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score

#--------------------Parameters---------------------#

# Change this to reduce cpu load
# Hey, if you have 32 cores, might as well use them :)
cores = mp.cpu_count()

#---------------------------------------------------#



# Given prepared datapoints, create a subset where categories have been removed
# This is to reduce redundant feature extraction work.
def subset_datapoints(X_train, X_val, y_train, y_val, n_species):
    classes = np.unique(np.concatenate((y_train, y_val)))
    if n_species >= len(classes):
        return X_train, X_val, y_train, y_val
    else:
        chosen_species = classes[:n_species]
        X_train_n = np.array([X_train[i] for i in range(len(y_train)) if␣
 ↪(y_train[i] in chosen_species)])
        X_val_n = np.array([X_val[i] for i in range(len(y_val)) if (y_val[i] in␣
 ↪chosen_species)])
        y_train_n = np.array([v for v in y_train if (v in chosen_species)])
        y_val_n = np.array([v for v in y_val if (v in chosen_species)])
        print(f'Set sizes train/validation: [{len(X_train_n)}, {len(X_val_n)}]')
        return X_train_n, X_val_n, y_train_n, y_val_n
```

```python
# If a method is worse than this,
# then we have definitely failed
def random_guess_accuracy(y_train, y_val):
    n_classes = len(np.unique(np.concatenate((y_train, y_val))))
    return 1.0 / n_classes


# Logistic learner for one dataset
def logistic_learn(input):

    X_train = input[0]
    X_val = input[1]
    y_train = input[2]
    y_val = input[3]

    model = LogisticRegression(solver='liblinear', max_iter=100)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_val)

    accuracy = accuracy_score(y_val, y_pred)
    rand_acc = random_guess_accuracy(y_train, y_val)

    # Use micro for multiclass scoring
    precision = precision_score(y_val, y_pred, average='micro')
    recall = recall_score(y_val, y_pred, average='micro')

    # Formula taken from scikit-learn.org
    f1 = 2 * (precision * recall) / (precision + recall)

    # Return metrics
    return np.array([
        accuracy,
        rand_acc,
        (accuracy-rand_acc),
        precision,
        recall,
        f1
    ])


# Train two logistic regressors
def double_trainer(input):
    return [logistic_learn(dataset) for dataset in input]
```

```
#-----------------Multiprocessing-------------------#

# Perform logistic learning accelerated by multiprocessing
# Call this function to spawn a child process that further
# distributes work among cpu cores
def parallel_learners(X_train, X_val, y_train, y_val, max_species, min_species,␣
 ↪step):

    print('Setting up logistic trainers for parallel execution: ')
    with Popen('python bird_parallel_learn.py', cwd=os.getcwd(), stdin=PIPE,␣
 ↪stdout=PIPE, universal_newlines=True) as p:

        # Send dataset
        with p.stdin as pipe:
            print('Transfering dataset to workers')
            pipe.write(f'{max_species}\n')
            pipe.write(f'{min_species}\n')
            pipe.write(f'{step}\n')
            pipe.write(f'{len(X_train)}\n')
            pipe.write(f'{len(X_val)}\n')
            pipe.write(f'{len(y_train)}\n')
            pipe.write(f'{len(y_val)}\n')
            pipe.flush()
            print(f'Set[{len(X_train)}, {len(X_val)}]')

            for vec in X_train:
                for v in vec:
                    pipe.write(f'{v} ')
                pipe.write('\n')
                pipe.flush()

            for vec in X_val:
                for v in vec:
                    pipe.write(f'{v} ')
                pipe.write('\n')
                pipe.flush()

            for v in y_train:
                pipe.write(f'{v}\n')
                pipe.flush()

            for v in y_val:
                pipe.write(f'{v}\n')
                pipe.flush()


        print(f'Initializing trainers: ')
        data = []

        # Receive progress updates
        for line in p.stdout:
            l = line.strip()
            print(l)
            if l == 'Done':
                break

        # Receive data
        for line in p.stdout:
            data.append(line.strip())

        # Reformat
        data_vectors = np.array([[float(v) for v in d.split(' ')] for d in␣
 ↪data])
        return data_vectors



# Worker process main function

if __name__ == '__main__':

    # Receive data for training

    max_species = int(sys.stdin.readline().strip())
    min_species = int(sys.stdin.readline().strip())
    step = int(sys.stdin.readline().strip())
    X_train_len = int(sys.stdin.readline().strip())
    X_val_len = int(sys.stdin.readline().strip())
    y_train_len = int(sys.stdin.readline().strip())
    y_val_len = int(sys.stdin.readline().strip())

    X_train = np.array([[float(v) for v in (sys.stdin.readline().strip().
 ↪split(' '))] for idx in range(X_train_len)])
    X_val = np.array([[float(v) for v in (sys.stdin.readline().strip().split('␣
 ↪'))] for idx in range(X_val_len)])
    y_train = np.array([int(sys.stdin.readline().strip()) for idx in␣
 ↪range(y_train_len)])
    y_val = np.array([int(sys.stdin.readline().strip()) for idx in␣
 ↪range(y_val_len)])

    # Create datasets for all species numbers
    datasets = [
        subset_datapoints(X_train, X_val, y_train, y_val, n_species)
        for n_species in range(min_species, max_species + 1, step)
    ]

    # These are ordered from shortest to longest, regroup based on time demand
    # [1, 2, 3, 4, 5, 6, 7] => [[1, 7], [2, 6], [3, 5], [4]]
    # [1, 2, 3, 4, 5, 6] => [[1, 6], [2, 5], [3, 4]]

    print('Partitioning data')

    N_datasets = len(datasets)

    doubled_sets = [[datasets[k], datasets[N_datasets - k - 1]] for k in␣
↪range(N_datasets // 2)]
    if N_datasets % 2 == 1: doubled_sets.append([datasets[N_datasets // 2]])

    completed = []

    print('Processing...')
    sys.stdout.flush()

    with mp.Pool(cores) as p:
        completed = p.map(double_trainer, doubled_sets)
    print('Done')

    # Reorder back
    output = [completed[k][0] for k in range(N_datasets // 2)]
    if N_datasets % 2 == 1: output.append(completed[N_datasets // 2][0])
    output.extend([completed[N_datasets // 2 - k - 1][1] for k in␣
↪range(N_datasets // 2)])

    # Send back data:
    for feature in output:
        vec = feature.flatten()
        for v in vec:
            print(v, end=' ')
        print()
        sys.stdout.flush()
```

bird_utility

October 9, 2022

```
[ ]: import numpy as np
     import librosa as lr
     import matplotlib.pyplot as plt
     from librosa.display import specshow
     from matplotlib.ticker import MaxNLocator





     # Plot audio in the time domain
     def audio_plot(data, sampling_rate, title='Audio'):
         fig, ax = plt.subplots()
         ax.xaxis.set_major_formatter(lambda tval, tpos : '%.1f' % (1.0 * tval /␣
      ↪sampling_rate))
         ax.xaxis.set_major_locator(MaxNLocator(integer=True))
         plt.xlabel("Time")
         plt.ylabel("Amplitude")
         ax.set_title(title)
         ax.plot(data)
         plt.show()



     # Plot spectrograms

     def spectral_power_plot(data, sampling_rate, title='Audio'):
         fig, ax = plt.subplots()
         img = specshow(lr.power_to_db(data, ref=np.max), x_axis='time',␣
      ↪y_axis='mel', sr=sampling_rate, ax=ax)
         fig.colorbar(img, ax=ax, format='%+2.0F dB')
         ax.set_title(title)
         plt.show()

     def spectral_amplitude_plot(data, sampling_rate, title='Audio'):
         fig, ax = plt.subplots()
```

```
    img = specshow(lr.amplitude_to_db(data, ref=np.max), x_axis='time',␣
 ↪y_axis='mel', sr=sampling_rate, ax=ax)
    fig.colorbar(img, ax=ax, format='%+2.0F dB')
    ax.set_title(title)
    plt.show()



# Special plots

def mfcc_plot(data, sampling_rate, title='Audio'):
    fig, ax = plt.subplots()
    img = specshow(data, x_axis='time', sr=sampling_rate, ax=ax)
    fig.colorbar(img, ax=ax, format='%+2.0F dB')
    ax.set_title(title)
    plt.show()

def centroid_plot(data, sampling_rate, title='Audio'):
    fig, ax = plt.subplots()
    times = lr.times_like(data, sr=sampling_rate)
    ax.xaxis.set_major_formatter(lambda tval, tpos : '%.1f' % (1.0 * tval /␣
 ↪sampling_rate))
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    ax.set_yscale('log')
    plt.xlabel("Time")
    plt.ylabel("Hz")
    ax.plot(times, data.T)
    ax.set_title(title)
    plt.show()
```