hybrid-vocal-classifier Documentation

Release 0.1.0

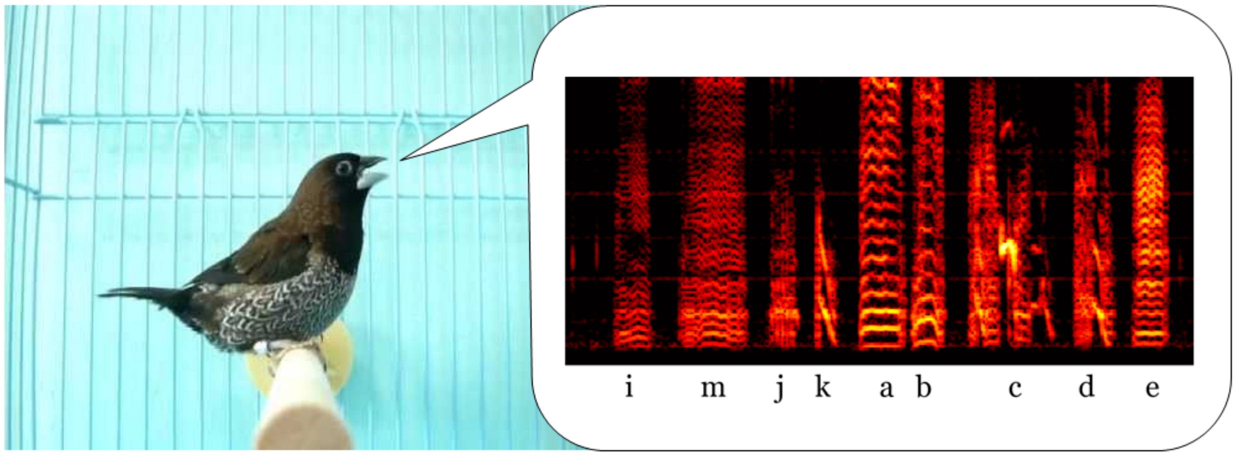
David Nicholson

Dec 27, 2021

Contents

CHAPTER 1

a Python machine learning library for animal vocalizations and bioacoustics



the `hybrid-vocal-classifier` library (`hvc` for short) makes it easier for researchers studying animal vocalizations and bioacoustics to apply machine learning algorithms to their data. Its focus on automating the sort of annotations often used by researchers studying vocal learning sets `hvc` apart from more general software tools for bioacoustics.

In addition to automating annotation of data, `hvc` aims to make it easy for you to compare different machine learning models that researchers have proposed, using the data you have in your lab, so you can see for yourself which one works best for your needs. A related goal is to help you figure out just how much data you have to label to get "good enough" accuracy for your analyses.

You can think of `hvc` as a high-level wrapper around the scikit-learn library, plus built-in functionality for working with annotated animal sounds.

Running `hvc` requires almost no coding. Users write simple Python scripts, and most will have to only adapt the examples from the documentation. Large batch jobs can be run with configuration files written in YAML, an easy-to-read format commonly used for configuration files. Again, most users will only have to copy the example `.yml` files and then change a couple of options to work with their own datasets.

This code sample gives a high-level view of how you run `hvc`:

```python
import hvc

# extract features from audio to train machine learning models
hvc.extract('extract_config.yml')  # using .yml config file
# train models/classifiers and select model with best accuracy
hvc.select('select_config.yml')
# use trained model to predict labels for unlabeled data
hvc.predict('predict_config.yml')
```

1.1 Advantages of hybrid-vocal-classifier

- frees up hundreds of hours spent annotating data by hand
- completely open source, free
- makes it easy to compare multiple machine learning algorithms
- almost no coding required, configurable with text files
- **built on top of Python packages road-tested by the greater data-science community:** numpy , scipy , matplotlib , scikit-learn , keras

1.2 Documentation

1.2.1 Tutorial

"autolabeling" with k-Nearest Neighbors

This tutorial will walk you through using `hvc` to automatically label Bengalese finch song with the k-Nearest Neighbors algorithm. We call this the "autolabel" workflow (for more detail after going through this tutorial, please see autolabel in the *workflows: how to work with hvc* section of *How-To Guides*.)

There's three main *modules* in `hvc` that you will use in the autolabel workflow: `extract` to extract features, `select` to select a model, and `predict` to predict labels for unlabeled data. The steps below walk you through doing that.

A convenient way to work through this tutorial would be in iPython, so you might first start iPython from the command line, like this:

```
(my-hvc-environment) $ ipython
```

iPython is not installed automatically with `hvc` so you'll need to install it. If you're using the `conda` package manager, this is as easy as:

```
(my-hvc-environment) $ conda install ipython
```

You can also use Jupyter notebooks from the tutorial here:
https://github.com/NickleDave/hybrid-vocal-classifier-tutorial

First you `import` the library so you can work with it.

0. Label a small set of songs to provide training data for the models, typically ~20-40 songs.

Here you would label your own song, using your software of choice (evsonganaly, Sound Analysis Pro, Praat) but for this example you can download some data that is already hand labeled from a repository.

1. Pick a machine learning algorithm/model and the features used to train the model.

In this case we use the k-Nearest Neighbors (k-NN) algorithm. This algorithm is quick to apply to data but at least one empirical study shows that it does not give the best accuracy on Bengalese finch song. You'll use the features built into the library that have been tested with k-NN. These features are based in part on those developed by the Troyer lab (http://www.utsa.edu/troyerlab/software.html).

You specify the models and features in a configuration file ("config" for short). More information about all the parameters in the config file can be found on the page :ref:`writing-extract-config`. For now you can just copy the text below and save it in some file. The config is written in YAML, a language for writing data structures (such as different types of variables in a programming language).

```
extract:
  spect_params:
    ref: evsonganaly
  segment_params:
    threshold: 1500 # arbitrary units of amplitude
    min_syl_dur: 0.01 # ms
    min_silent_dur: 0.006 # ms

  todo_list:
    -
      bird_ID : gy6or6
      file_format: evtaf
      feature_group:
        - knn
      data_dirs:
        - .\gy6or6\032612

      output_dir: .\gy6or6\

      labelset: iabcdefghjk
```

2. Extract features for that model from song files that will be used to train the model.

You call the `extract` module and pass it the name of the `yaml` config file as an argument. In the example below, the config file was saved as `'gy6or6_autolabel_example.knn.extract.config.yml'`.

3. Pick the hyperparameters used by the algorithm as it trains the model on the data.

Now we use a convenience function to get an estimate of what value for our **hyperparameters** will give us the best accuracy when we train our machine learning models. The k-Nearest Neighbors algorithm has one main hyperparameter, the number of neighbors k in feature space that we look at to determine the label for a new syllable we are trying to classify.

4. Train, i.e., fit the model to the data

5. Select the best model based on some measure of accuracy.

Again we use a config file. In the config file, we specify the name of the feature file saved by `hvc.extract`. Again you can just copy and paste the text below.

The key things to modify here are the hyperparameter :math:‘k‘ and the name of the feature file. You will choose the value for :math:‘k‘ based on your results from running ‘‘hvc.utils.find_best_k‘‘. You will get the name of the feature file from the directory created when you ran ‘‘hvc.extract‘‘. The name of the directory will be something like ‘‘extract_output_bird_ID_date‘‘. Make sure that on the line that says ‘‘feature_file:‘‘, you paste the name of the feature file after the colon. The name will have a format like ‘‘summary_file_bird_ID_date‘‘.

```
select:

  num_replicates: 10
  num_train_samples:
    start : 50
    stop : 250
    step : 50
  num_test_samples: 500

  models:
    -
      model_name: knn
      feature_list_indices: [0,1,2,3,4,5,6,7,8]
      hyperparameters:
        k : 4

  todo_list:
    - #1
      feature_file: .\gy6or6\extract_output_171031_214453\summary_feature_file_
↪created_171031_214642
      output_dir: .\gy6or6\
```

Now you can use `hvc.select` to select the best model. `hvc.select` takes the name of the config file as an argument, which in this example is `gy6or6_autolabel.example.select.knn.config.yml`.

6. Using the fit model, predict labels for unlabeled data.

Here you also use a config file.

** The key things to modify here is the `model_meta_file` parameter. `hvc.select` will also have created a directory, and for each model it fit, it will have saved two files, a `.model` file and a `.meta` file. The `.meta` file contains all the metadata that `hvc` needs to be able to use the `.model` file. You choose whichever `.meta` file gave you the best results according to the metric you're using, e.g. the default of average accuracy across syllable classes. You also need to specify the directories with unlabeled data, under the `data_dirs` section.**

```
predict:
  todo_list:
    -
      bird_ID : gy6or6
      file_format: evtaf
      data_dirs:
        - C:\Users\Seymour Snyder\Documents\example_song\032612
      model_meta_file: .\gy6or6\select_output_171031_215004\knn_k4\knn_200samples_
↪replicate9.meta
```

(continues on next page)

(continued from previous page)

```
      output_dir: .\gy6or6
      predict_proba: True
      convert: notmat
```

1. In a text editor, open
2. On the line that says `model_meta_file:`, after the colon, paste the name of a meta file from the `select` output. The name will have a format like `summary_file_bird_ID_date`.
3. Below the line that says `data_dirs:`, after the dash, add the path to the other folder of data that you downloaded.

Lastly you use the `hvc.predict` module to predict labels for new syllables. `hvc.predict` also takes a config file name as an argument. In this example the file name is `gy6or6_autolabel.example.knn.predict.config.yml`.

```
parsed predict config
Changing to data directory: C:/Data/gy6or6_all_files/032612
Processing audio file 1 of 39.
Processing audio file 2 of 39.
...
Processing audio file 39 of 39.
predicting labels for features in file: features_from_032612_created_171206_013759
converting to .not.mat files
```

Congratulations! You have auto-labeled an entire day's worth of data in just a few minutes!

1.2.2 How-To Guides

This section provides more detailed "how-to" guides to consult.

workflows: how to work with `hvc`

There are two main workflows for using `hvc`. Click on the links below for a high-level overview of each:

1. autolabel: for researchers that want to automate labeling of vocalizations.
2. autocompare: for researchers that want to compare different machine learning algorithms.

how to write YAML configuration files

Pages here explain in detail how to write YAML configuration files.

how to write yaml files used by the *select* module

As described in the introduction, a crucial step in using hybrid vocal classifier is selecting which models to use. This can be done in an automated way using the *select* module. Like the *extract* and *predict* modules, the *select* module works by parsing configuration files. Below the steps are outlined in writing the configuration files in yaml format.

what the select module gets out of the config file: models and data

There are two required elements in a select config file, that

correspond to the two main things that the *select* module needs to know:

1. *models*: what models to test. A Python list of dictionaries, as described below.
2. *todo_list*: where the data is to train and test those models. Another Python list of dictionaries, also described below.

The parser that parses the *select* config file is written so that you don't have to repeat yourself. You can put one *models* list at the top of the file, and then for each dataset in the *todo_list*, the *select* module will train and test all the models that are specified in that top-level *models* list. Like so:

..include

However you can also define a *models* dictionary for each *todo_list*, in case you need to test different models for different datasets, and want to run them all from one script.

..include

the *models* list

To be parsed correctly, the *models* list needs to have the right structure. In yaml terminology, this is a list. Once parsed into Python, it becomes a list of dictionaries. For that reason the structure is described in terms of the keys and values required for each dictionary. Each dictionary in the list represents one model that the *select* module will test. There are a couple of required keys for each model dictionary.

required key 1: hyperparameters

These models are found using machine learning algorithms. A model can be thought of as a function with parameters, like the beta terms of a linear regression. To find these parameters, the algorithm must train on the data, and this training also has parameters, for example the number of neighbors used by the K-nearest neighbor algorithm. These parameters of the algorithm are known as **hyperparameters** to distinguish them from the parameters found by the algorithm.

the *todo list*

1.2.3 Reference

yaml config

spec for yaml config files

This document specifies the structure of HVC config files written in yaml. It is a painfully dry document that exists to guide the project code, not to teach someone how to write HVC config files. For a gentle introduction to writing config files, please see the writing_config_files.

Essentially, each config file specifies a list of *jobs*. Each *job* in a list will typically correspond to data files from one bird.

Config files consist of three sections:

1. *global_config*: parameters that apply to all *jobs*

2. *model_selection*: list of *jobs* for selecting machine learning models
3. *prediction*: list of *jobs* that apply models to unclassified data

## global_config As the name implies, parameters in the *global_config* section apply to all jobs. The *global_config* is a dictionary of dictionaries.

Example: ‘‘‘ yaml global_config:

spect_params : samp_freq : 32000 # Hz window_size : 512 window_step : 32 freq_cutoffs : [1000,8000]

neural_net : syl_spect_width : 300

‘‘‘

model_selection

***model_selection* is a list of *jobs*. Each *job* is a dictionary.** Hence *model_selection* is a list of dictionaries.

Each *job*, i.e. each item in the list, is marked with an empty dash. Below each empty dash appear the keys and values that make up the dictionary.

A *job* in the ‘model_selection‘ section **must** include the following keys:

- *bird_ID* : string, alphanumeric, identifies bird
- *train* : dictionary with parameters for training dataset
- *test* : dictionary with parameters for testing dataset - both *train* and *test* contain a list *dirs*. Each item in *dirs*

  is a string, and that string **must** be a path to a directory of audio files (expected to contain song from the bird *bird_ID*).
- *output_dir* : string, directory where output will be saved. HVC

creates a new subfolder in the given directory. - *labelset* : string, labels used for syllabes. Only syllables with the labels in *labelset* will be included in the training and testing

datasets.

**If a parameter is defined in *global_config* and then defined again in a *job*, the value defined in the *job* takes precedence over the

global_config value, but only for that job.**

Example: ‘‘‘ yaml model_selection: # list of dictionaries, dash without key next to is a list item so each dictionary is an item in the list

- **# i.e. this is dictionary 1** bird_ID : gr41rd51

  train :

  dirs:

  – C:DATAgr41rd51pre_surgery_baseline06-21-12

  test :

  dirs:

  – C:DATAgr41rd51pre_surgery_baseline06-19-12

  – C:DATAgr41rd51pre_surgery_baseline06-20-12

        - C:DATAgr41rd51pre_surgery_baseline06-22-12

    output_dir: C:DATAgr41rd51

    labelset : iabcdefgjkm

    spect_params [# not required, but will take precedence over spect_params in global_config] samp_freq : 32000 # Hz window_size : 512 window_step : 32 freq_cutoffs : [1000,10000]

‘‘‘

prediction

Like *model_selection*, the *prediction* section is a list of *job* dictionaries.

A *job* in the ‘prediction‘ section must include the following keys:

- *bird_ID* : string, alphanumeric, identifies bird
- *model_file* : string, a file name. Either a scikit-learn model that

has been ‘pickle‘d or ‘dump‘ed by joblib, or an hdf5 model output by Keras.

‘‘‘ yaml prediction:

- bird_ID : gr41rd51 model_file : gr41rd51_svm.pkl

‘‘‘

parameters

The parameters listed below can appear in either *global_config* or a *job*.

- **spect_params :**
    - samp_freq : integer
    - window_size : integer
    - window_step : integer
    - freq_cutoffs : list
- **num_train_songs :**
    - start : integer
    - stop : integer
    - step : integer
- **num_train_samples :**
    - start : integer
    - stop : integer
    - step : integer
- **models :**
    - knn
    - linsvm
    - svm

  - neural_net

spec for YAML files to configure feature extraction

This document specifies the structure of HVC config files written in YAML.

structure

Every `extract.config.yml` file should be written in YAML as a dictionary with (key, value) pairs. In other words, any YAML file that contains a configuration for feature extraction should define a dictionary named `extract` with keys as outlined below.

required key: `todo_list`

Every `extract.config.yml` file has exactly one required key at the top level:

`todo_list`: list of dicts list where each element is a dict. each dict sets parameters for a 'job', typically data associated with one set of vocalizations.

optional keys

`extract.config.yml` files *may* optionally define two other keys at the same level as `todo_list`. Those keys are `spect_params` and `segment_params`. As might be expected, `spect_params` is a dict that contains parameters for making spectrograms. The `segment_params` dict contains parameters for segmenting song. Specifications for these dictionaries are given below.

When defined at the same level as `todo_list` they are considered "default" and apply to all items in the list. If an element in *todo_list* defines different values for any of these keys, the value assigned in that element takes precedence over the *default* value.

specification for dictionaries in `todo_list`

required keys

Every dict in a `todo_list` has the following required keys:

- `bird_ID` : str for example, `bl26lb16`
- `file_format`: str one of `{'evtaf','koumura'}`
- `data_dirs`: list of str directories containing data each str must be a valid directory that can be found on the path for example

```
- C:\DATA\bl26lb16\pre_surgery_baseline\041912
- C:\DATA\bl26lb16\pre_surgery_baseline\042012
```

- `output_dir`: str directory in which to save output if it doesn't exist, HVC will create it for example, `C:\DATA\bl26lb16\`
- `labelset`: str string of labels corresponding to labeled segments from which features should be extracted. Segments with labels not in this str will be ignored. Converted to a list but not necessary to enter as a list. For example, `iabcdef`

Finally, each dict in a `todo_list` must define *either* `feature_list` *or* a `feature_group`

- **`feature_list`** [list] named features. See the list of named features here: named_features
- **`feature_group`** [str or list] named group of features, list if more than one group one of `{'knn','svm'}`
- Note that a `todo_list` can define *both* a `feature_list` and a `feature_group`. In this case features from the `feature_group` are added to the `feature_list`.

Additional variables are added to the feature files that are output by `featureextract.extract` to keep track of which features belong to which feature group.

specification for `spect_params` and `segment_params` dictionaries

- **`spect_params`: dict** parameters to calculate spectrogram keys correspond to parameters/arguments passed to Spectrogram class for __init__. **must** have *either* a `ref` key *or* the `nperseg` and `noverlap` keys as defined below:

  `ref` [str] one of `{'tachibana','koumura'}` Use spectrogram parameters from a reference. `'tachibana'` uses spectrogram parameters from[1], `'koumura'` uses spectrogram parameters from[2].

  `nperseg` [int] numper of samples per segment for FFT, e.g. 512

  `noverlap` [int] number of overlapping samples in each segment

  the following keys are all optional for `spect_params`:

  `freq_cutoffs` [two-element list of integers] limits of frequency band to keep, e.g. [1000,8000] `Spectrogram.make` keeps the band:

  freq_cutoffs[0] >= spectrogram > freq_cutoffs[1]

  `window` [str] window to apply to segments valid strings are `'Hann', 'dpss', None` Hann – Uses `np.Hanning` with parameter `M` (window width) set to value of `nperseg` dpss – Discrete prolate spheroidal sequence AKA Slepian.

  Uses `scipy.signal.slepian` with M parameter equal to `nperseg` and width parameter equal to `4/nperseg`, as in[2].

  `filter_func` [str] filter to apply to raw audio. valid strings are `'diff'` or `None` `'diff'` – differential filter, literally `np.diff` applied to signal as in[1]. `None` – no filter, this is the default

  `spect_func` [str] which function to use for spectrogram. valid strings are ‘scipy’ or ‘mpl’. `'scipy'` uses `scipy.signal.spectrogram`, `'mpl'` uses `matplotlib.matlab.specgram`. Default is `'scipy'`.

  `log_transform_spect` [bool] if True, applies np.log10 to spectrogram to increase range. Default is True.

`segment_params`: dict parameters for dividing audio into segments, defined below with the following keys

`threshold` [int] value above which amplitude is considered part of a segment. default is 5000.

`min_syl_dur` [float] minimum duration of a segment. default is 0.02, i.e. 20 ms.

[1] Tachibana, Ryosuke O., Naoya Oosugi, and Kazuo Okanoya. “Semi-

[2] Koumura, Takuya, and Kazuo Okanoya. “Automatic recognition of element

min_silent_dur [float] minimum duration of silent gap between segment. default is 0.002, i.e. 2 ms.

example extract.config.yml files

These are some of the `extract.config.yml` files used for testing, found in `hybrid-vocal-classifier/tests//data_for_tests/config.yml/`:

```
extract:
  spect_params:
    ref: tachibana
  segment_params:
    threshold: 1500 # arbitrary units of amplitude
    min_syl_dur: 0.01 # ms
    min_silent_dur: 0.006 # ms

  todo_list:
    -
      bird_ID : gy6or6
      file_format: cbin
      feature_group:
        - knn
      data_dirs:
        - ../cbins/gy6or6/032312
        - ../cbins/gy6or6/032412

      output_dir: replace with tmp_output_dir

      labels_to_use: iabcdefghjk
```

```
extract:
  spect_params:
    ref: tachibana
  segment_params:
    threshold: 1500 # arbitrary units of amplitude
    min_syl_dur: 0.01 # ms
    min_silent_dur: 0.006 # ms

  todo_list:
    -
      bird_ID : gy6or6
      file_format: cbin
      feature_group:
        - svm
      data_dirs:
        - ../cbins/gy6or6/032312
        - ../cbins/gy6or6/032412

      output_dir: replace with tmp_output_dir

      labels_to_use: iabcdefghjk
```

```
extract:
  spect_params:
    ref: koumura
  segment_params:
```

(continues on next page)

(continued from previous page)

```
    threshold: 1500 # arbitrary units of amplitude
    min_syl_dur: 0.01 # ms
    min_silent_dur: 0.006 # ms

  todo_list:
    -
      bird_ID : gy6or6
      file_format: cbin
      feature_list:
        - flatwindow
      data_dirs:
        - ../cbins/gy6or6/032312
        - ../cbins/gy6or6/032412

      output_dir: replace with tmp_output_dir

      labels_to_use: iabcdefghjk
```

references

automatic classification of birdsong elements using a linear support vector machine." PloS one 9.3 (2014): e92584.

classes and boundaries in the birdsong with variable sequences." PloS one 11.7 (2016): e0159188.

spec for YAML files to configure model selection

This document specifies the structure of HVC config files written in YAML.

structure

Every `select.config.yml` file should be written in YAML as a dictionary with (key, value) pairs. In other words, any YAML file that contains a configuration for model selection should define a dictionary named `select` with keys as outlined below.

required key: `todo_list`

Every `select.config.yml` file has exactly one required key at the top level:

`todo_list`: list of dicts list where each element is a dict. each dict sets parameters for a 'job', typically data associated with one set of vocalizations.

optional keys

`select.config.yml` files *may* optionally define other keys at the same level as `todo_list`. Those keys are:

`num_replicates`: int number of replicates, i.e. number of folds for cross-validation

`num_test_samples`: int number of samples from feature file to put in testing set

`num_train_samples`: int number of samples from feature file to put in training set

models: list list of dictionaries that define models to be tested on features

When defined at the same level as `todo_list` they are considered `default`. If an element in `todo_list` defines different values for any of these keys, the value assigned in that element takes precedence over the `default` value.

specification for dictionaries in todo_list

required keys

Every dict in a `todo_list` has the following required keys:

- `feature_file`: str for example: `C:\Data\gy6or6\extract_output_170711_0104\summary_feature_f`
- `output_dir`: str path to directory in which to save output if it doesn't exist, HVC will create it for example, `C:\DATA\bl26lb16\`

optional keys

As stated above, these can all be defined at the top level of the file. If they are also defined for any dict in a `todo_list`, then that definition will override the top-level definition.

- **`models`: list of dicts** dictionary of models, as defined below. Required if not defined at top level of file.
- **`num_replicates`: int** number of replicates, i.e. number of folds for cross-validation
- **`num_test_samples`: int** number of samples from feature file to put in testing set
- **`num_train_samples`: int** number of samples from feature file to put in training set

specification for models list of dicts

Every dict in a `models` list has the following required keys:

- **`model_name`: str** name of model, e.g. 'svm'
- **`hyperparameters`: dict** with hyperparameters defined for each model

Every dict in a `models` list must also specify the features with which to train the model. One of the following is valid, as specified in `validation.yml`.

- **`feature_list_indices`: list of ints** corresponding to elements in list of feature names in feature_file e.g., `[0,1,2,5,7]`
- **`feature_group`: str** name of a feature group: one of `{'knn','svm'}`
- **`neuralnet_input`: str** name of input for am artificial neural net: `{'flatwindow'}`

example `select_config.yml`

These are some of the `select.config.yml` files used for testing, found in `hybrid-vocal-classifier/tests/data_for_tests/config.yml/`:

spec for YAML files to configure label prediction

This document specifies the structure of HVC config files written in YAML.

structure

Every `predict.config.yml` file should be written in YAML as a dictionary with (key, value) pairs In other words, any YAML file that contains a configuration for feature extraction should define a dictionary named 'predict' with keys as outlined below.

required key: `todo_list`

Every `predict.config.yml` file has exactly one required key at the top level:

`todo_list`: list of dicts list where each element is a dict. each dict sets parameters for a 'job', typically data associated with one set of vocalizations.

specification for dictionaries in `todo_list`

required keys

Every dict in a *todo_list* has the following required keys:

- `bird_ID` : str for example, *bl26lb16*
- `file_format`: str {'evtaf','koumura'}
- `data_dirs`: list of str directories containing data each str must be a valid directory that can be found on the path for example

```
- C:\DATA\bl26lb16\pre_surgery_baseline\041912
- C:\DATA\bl26lb16\pre_surgery_baseline\042012
```

- `model_file`: str filename of machine learning model / neural network that will be used to predict labels for syllables example: *somedir/select_output_170814_005430/knn/knn_100samples_replicate0*
- `output_dir`: str directory in which to save output if it doesn't exist, HVC will create it for example, `C:\DATA\bl26lb16\`
- `predict_proba` : bool If True, calculate probabilities for predicted labels.

example `predict_config.yml`

These are some of the `predict.config.yml` files used for testing, found in `hybrid-vocal-classifier/tests/data_for_tests/config.yml/`:

Features

This section contains information on engineered features that `hvc` provides for machine learning models.

named features

These features are pre-defined and can be referred to by name in the *feature_list* of YAML files for *extract*.

feature group *Tachibana*:

- *mean_spectrum*
- *mean_delta_spectrum* : 5-order delta of spectrum
- *mean_cepstrum*
- *mean_delta_cepstrum* : 5-order delta of cepstrum
- *dur* : duration
- *mA* : additional subset of features listed below
- *SpecCentroid*
- *SpecSpread*
- *SpecSkewness*
- *SpecKurtosis*
- *SpecFlatness*
- *SpecSlope*
- *Pitch*
- *PitchGoodness*
- *Amp*

References

automatic classification of birdsong elements using a linear support vector machine." PloS one 9.3 (2014): e92584.

models

k-Nearest Neighbors (kNN)

API reference

hvc

hvc package

Subpackages

hvc.features package

Submodules

hvc.features.extract module

hvc.features.knn module

hvc.features.tachibana module

Module contents

hvc.neuralnet package

Submodules

hvc.neuralnet.conv_models module

hvc.neuralnet.models module

Module contents

hvc.parse package

Submodules

hvc.parse.extract module

hvc.parse.predict module

hvc.parse.select module

Module contents

Submodules

hvc.audiofileIO module

hvc.evfuncs module

hvc.featureextract module

hvc.koumura module

hvc.labelpredict module

hvc.metrics module

hvc.modelselect module

hvc.parseconfig module

hvc.randomdotorg module

hvc.utils module

Module contents

1.2.4 Development

This section provides more detail for developers, including:

- Descriptions of how the code base works under the hood
- a roadmap for the project

code base

These pages contain information on the code base and its design, as a reference for development, including pseudocode-like descriptions of functions to orient someone reading the code.

# HVC workflow in detail This document explains in detail how functions and modules work, mainly as a reference for developers.

Take the example code from the intro notes page: ```Python import hvc

hvc.extract('extract_config.yml') hvc.select('select_config.yml') hvc.predict('predict_config.yml') ```

Here's a step-by-step outline of what happens under the hood: - *import hvc*

- automatically imports *featureextract*, *labelpredict*, and

modelselect **modules**

- specifically, the *extract*, *predict*, and *select* functions

  from their respective modules

- *hvc.extract('extract_config.yml')*
- first *parse.extract* parses the config file
- for each element in *todo_list* from config + for each data directory *datadir* in *todo_list*:
  - change to that directory
  - get all the audio files in that directory with *glob*
  - for each audo file: + run *features.extract.from_file* + add extracted features to *features_from_all_files*
- save all features in an output file
- *hvc.select('select_config.yml')*
- *hvc.predict('predict_config.yml')*

parse package, extract module

The extract module parses *extract.config.yml* files.

Here's a rough outline of how it works.

parseextract.py contains the following functions: - *validate_spect_params* - *validate_segment_params* - *_validate_feature_group_and_convert_to_list* - *_validate_todo_list_dict* - *validate_yaml*

validate_yaml

validate_yaml is the main function; it gets called by parse.

_validate_todo_list_dict

_validate_todo_list_dict is called by *validate_yaml* when it finds a key *todo_list* whose value is a list of dicts.

_validate_feature_group_and_convert_to_list

This function validates feature groups, which are validated differently than feature lists. Feature lists are validated by making sure every element in the list is a string found within a valid features list, which is a concatenation of all the features listend in *feature_groups.yml* in the parse module.

The parsing of a *feature_group* key is a little more complicated. The first step is to make sure the group or groups appear in the dictionary of valid feature groups in *hvc/parse/feature_groups.yml'. The keys of the dictionary of valid feature groups are the valid feature group names, and the values of the dictionary are the actual lists of features. If each 'str* in **'**feature_group1 is s a valid feature group, then the list of features is taken from the dictionary

of valid feature groups. The list is then validated by comparing it to the list of all features in *features.yml*.

This is to make sure the developer didn't make a typo. If *feature_group* is a list of feature group names, then *feature_list* will consist of all features from all groups in the list. A vector of the same length as the new feature list has values that indicate which feature group each element in the new feature list belongs to. This vector is named *feature_list_group_ID*. A dict named *ftr_group_dict* is also returned,

where each key is a name of a feature group and its

corresponding value is the ID number given to that feature group. Using this dict and the identity array, *hvc/parse/select* can pull the correct features out of a feature array given a feature group name.

Example: **'''**Python >>> ftr_tuple = _validate_feature_group_and_convert_to_list(feature_group=['knn','svm'])

```
>>> ftr_tuple[0]
```

['duration group','preceding syllable duration' . . .] # and so on

```
>>> ftr_tuple[1]
```

np.ndarray([0,0,0,0,0,1,1,1,1,1]) # some array with one of two ID numbers

```
>>> len(ftr_type[0]) = ftr_tuple[1].shape[-1]
```

True

```
>>> ftr_tuple[2]
```

{'knn': 0, 'svm': 1} **'''**

If a *feature_list* was passed to this function along with *feature_group*, the features from the feature groups are appended to the *feature_list*, and in the *feature_group_ID* vector, the original features from the original feature list have a value of *None*.

development roadmap

Here's a list of features with rough timelines.

immediate priority

- parsers for different file formats
- automate build
- mostly complete test coverage

not as immediate priority

bells and whistles

1.3 Installation

see install

1.4 Support

If you are having issues, please let us know.
Please post bugs on the Issue Tracker:
https://github.com/NickleDave/hybrid-vocal-classifier/issues
And please ask questions in the users' group:
https://groups.google.com/forum/?hl=en#!forum/hvc-users/join

1.5 Contribute

- Issue Tracker: https://github.com/NickleDave/hybrid-vocal-classifier/issues
- Source Code: https://github.com/NickleDave/hybrid-vocal-classifier/

1.6 License

BSD license.

1.7 Citations, repositories, and related work

If you use this library, please cite its DOI:

To cite the algorithms used, please see the listing in citations.
A list of repositories of birdsong is here: repos
A list of related works is here: related

To suggest or contribute algorithms or repositories:

Please feel free to start an issue on the Github repository
https://github.com/NickleDave/hybrid-vocal-classifier/issues
or comment in the users' group:
https://groups.google.com/forum/?hl=en#!forum/hvc-users/join

1.8 Code of Conduct

We welcome contributions to the codebase and the documentation, and are happy to help first-time contributors through the process. Project maintainers and contributors are expected to uphold the code of conduct described here: code-of-conduct

CHAPTER 2

Backstory

`hvc` was originally developed in the Sober lab as a tool to automate annotation of birdsong (as shown in the picture above). It grew out of a submission to the SciPy 2016 conference and later developed into a library, as presented in this talk: https://youtu.be/BwNeVNou9-s