Improved Analysis of Harbour Porpoise Sounds

Final Report

David K. Mellinger and Selene Fregosi

December 4, 2020

1. Introduction

The overarching goal of this effort is to provide information for assuring compliance with the Species at Risk Act (SARA) for tidal energy devices installed in the Bay of Fundy, Nova Scotia, Canada. Part of that goal is to monitor one of those species at risk, the harbour porpoise (*Phocoena phocoena*), to assess whether the devices are having any impact on the population. To do this, it is necessary to collect information on the occurrence of the porpoises over time, which is done by making underwater acoustic recordings and finding out how much of the time the porpoises are present. These recordings are sufficiently extensive, typically spanning months, that manual review is not practicable. An automated method is needed to scan the recordings for sounds of harbour porpoises and report the results in a way that can be used for SARA compliance.

The goal of this project was to build such an automated system. The approach taken was to construct a multi-stage system that takes as input a collection of recorded sound files and produces as output a series of tables and figures summarizing harbour porpoise presence found from the recordings. The structure of this was a system called *FindPorpoises* (Fig. 1) for automated detection and classification of harbour porpoise echolocation clicks, followed by a module for producing the desired tables and figures summarizing harbour porpoise presence. *FindPorpoises* in turn comprises three internal stages, for click detection, click classification, and click grouping.

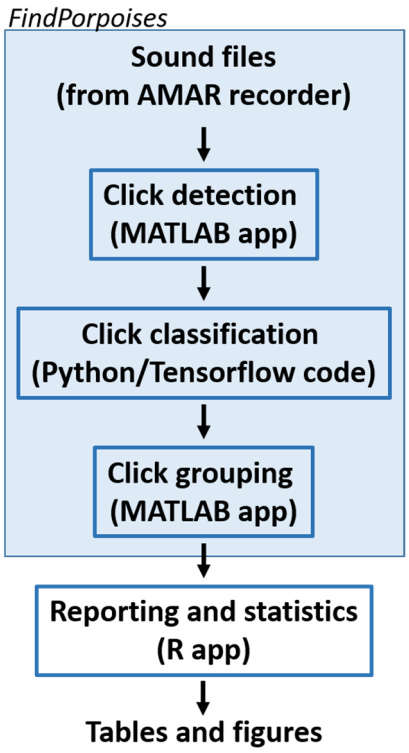


Figure 1. Overall structure of the system. The detection and classification portion is a single runnable system called 'FindPorpoises', which is followed by a step that is executed separately that makes tables and figures summarizing harbour porpoise presence.

2. Detector

The detector used was a based on the ratio of the energy in the porpoise frequency band to the energy in a lower "guard" band. The purpose of the guard band is to prevent loud noises that span the entire frequency range, such as clunking and snapping sounds, from triggering false detections. The detector was designed using the *Ishmael* (Fig. 2), a system for viewing, analyzing, and detecting bioacoustic sounds of interest (Mellinger et al. 2018). See Appendix A for the parameter values that control detector operation and Appendix C for details of the code implementing the detector.

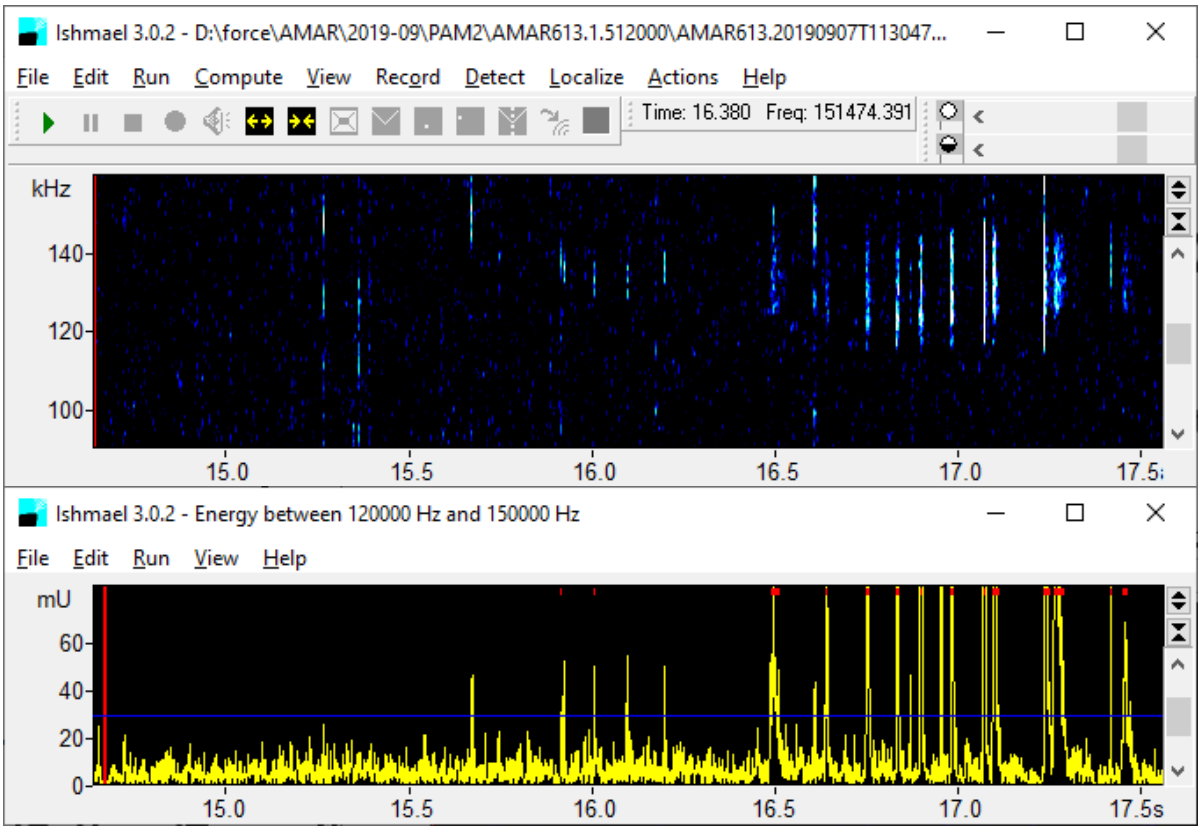


Figure 2. Energy detector running in Ishmael. The top panel shows a noise-equalized spectrogram including harbour porpoise echolocation clicks (vertical blue-to-white stripes); the bottom panel shows the detection function, which has a peak at times when harbour porpoises are present. The blue line is the detection threshold.

The detector operated on noise-equalized spectrograms. Noise equalization was performed to (1) make the background noise level relatively constant and small so that the energy ratio would not trigger on it (this was especially important in a place like Minas Passage, where flow rates and hence noise levels vary by several orders of magnitude); (2) reduce and eliminate long-duration stationary signals, such as from mechanical sources like vessels and machinery; and (3) correct for any non-flat frequency response in the hydrophone and data-acquisition system.

Detection parameters were tuned using sound files from FORCE known to have harbour porpoise clicks in them. Many of the recordings had quite high noise levels because they were made when the tide was flooding or ebbing strongly, which made detection challenging. A key detection parameter is the

detection threshold; setting it higher results in fewer false detections but more missed detections, and vice versa for setting it lower. Here, a relatively low threshold was chosen so as to have few missed detections, with the knowledge that the large number of false detections would be eliminated by the succeeding classifier stage. The complete list of detection parameters is in Appendix A.

The detector was initially operated in *Ishmael*, but when building the complete *FindPorpoises* system it was realized that having a separate piece of software to install, run, and maintain was not necessary when the computational process it was performing was a simple energy detector. Accordingly, the detector was made into MATLAB code and incorporated into *FindPorpoises*.

3. Classifier

The classifier was a part of the porpoise monitoring software that was critical to successful operation of the system. The classifier was a deep-learning system built using TensorFlow, a tool from Google, Inc. for training and using deep-learning systems. The front-end environment used to interface to TensorFlow was Jupyter Notebooks (https://jupyter.org) running Python (version 3.7) code. Developing and using a classifier consists of the steps of data preparation, training, and deployment.

Data preparation. A classifier is intended to assign its input data to one of several classes. In this case, the task was to train a classifier to recognize echolocation clicks from harbour porpoises, and the corresponding classes were called “Click” and “NoClick” for data with and without echolocation clicks. The most straightforward method of preparing data for a classifier is to have a number of labeled data instances that contain the all classes the classifier is being trained on so that it can learn the difference between the classes. In this case, that meant preparing some data with porpoise clicks labeled *Click*, as well as some data without clicks labeled *NoClick*. To prepare a data set with these labeled data instances, we examined the Minas Passage AMAR recordings in the spectrogram viewer Osprey (Mellinger 2014) and labeled times at which clicks occurred. Clicks were identifiable, usually readily so, in the spectrograms (Fig. 3). The labels consisted simply of times at which porpoise clicks occurred, expressed as times in seconds from the start of each sound file. A total 6,407 clicks from 22 sound files on 10 days, spanning times of heavy and light tidal flow noise, were manually labeled.

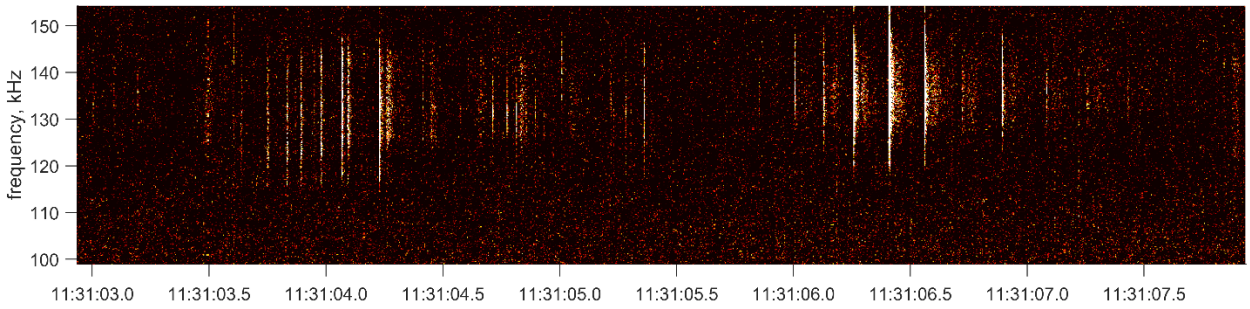


Figure 3. Clicks of harbour porpoises recorded by AMAR613 in Minas Passage on 7 Sept 2019.

Training data for the classifier was then generated from the labels and sound recordings. Each element of *Click* training data – the input to the classifier – consisted of a ½-second-long spectrogram between the frequencies of 90 and 160 kHz, equalized to a uniform background level. These spectrograms are stored as .png files, with each spectrogram 141x499 pixels (frequency x time); spectrograms of this size are the data inputs to the classifier network. Each sample had one of the labeled clicks at its center.

Training data files for the *NoClick* samples were generated using a set of sound files that were checked to ensure there were no porpoise clicks present. There were 59 of these "no clicks present" sound files representing 118 minutes of sound data from different points in the tide cycle so as to capture different noise conditions. The *NoClick* samples were also ½-second-long spectrograms, identical in shape to the *Click* samples. The *NoClick* samples were taken from times within these "no clicks present" files at which false detections occurred, with each false detection centered within its spectrogram. Because there was a need for tens of thousands of training samples but only 6,407 labeled clicks, data augmentation was used to expand the size of the data set. Augmentation was done by mixing the ½-second audio signal of a given click with a ½-second noise sample taken from the "no clicks present" sound files.

Training data for the *Click* and *NoClick* data sets was generated by the MATLAB routine `generateAugmentedDataset.m`. A total of 20,000 *Click* and 20,000 *NoClick* training data samples were generated. Figure 4 shows examples of *Click* and *NoClick* samples.

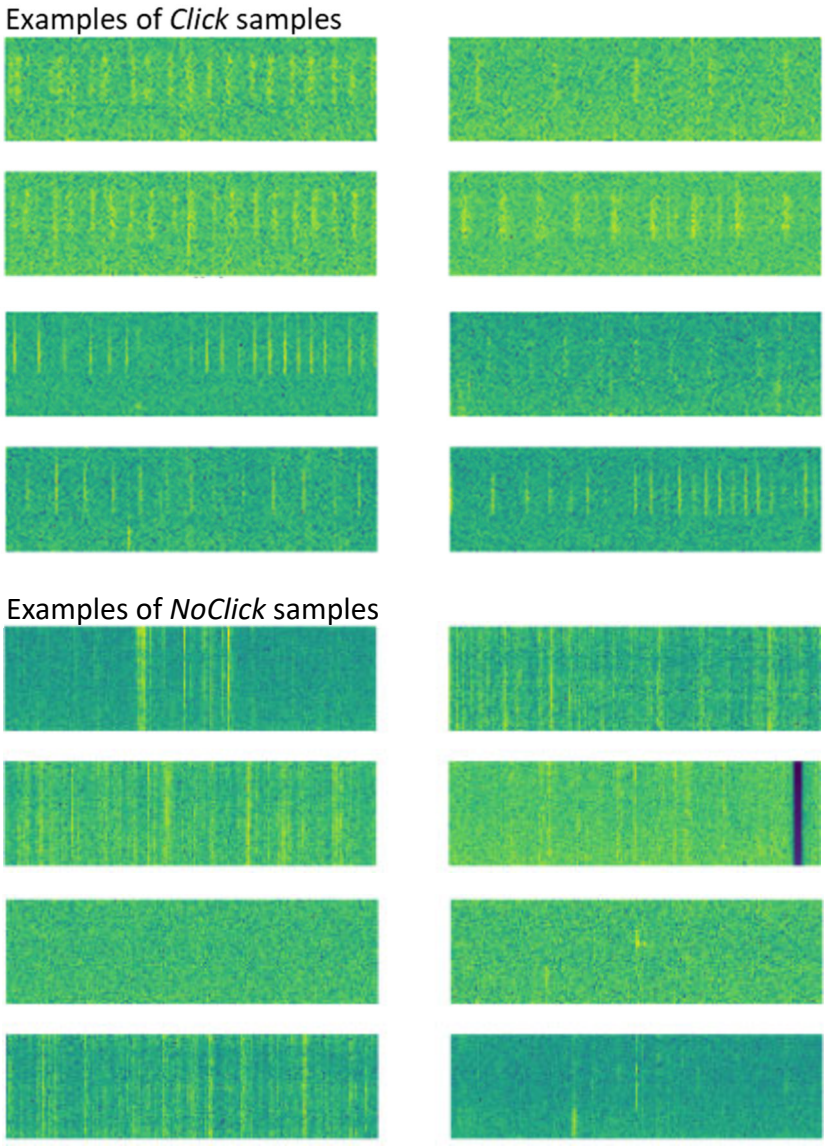


Figure 4. Examples of 'Click' and 'NoClick' spectrogram samples.

Training. Training using TensorFlow was done using Jupyter Notebooks. A variety of network architectures were tested, including one based on the Inception pre-trained network, and convolutional networks employing rectified linear units (ReLU) and pooling layers. A number of different network (model) shapes were tried as well, with different numbers of layers, layer sizes, and training epochs. Training using the prepared data was done by using 90% of the data (36,000 data samples, half *Click* and half *NoClick*) for training and using the remaining 10% (4,000 samples, likewise half and half) for testing to find the network's accuracy.

Network training was done in a series of 5-15 epochs depending on the network used and the convergence rate of training. The best-performing network ended up having four pairs of convolution-pooling layers and a 512-element flat, fully-connected (dense) layer before output (Fig. 5). This network architecture is defined in the Jupyter Notebook `porp_try9_FINAL.ipynb` and also in `classify_grams.py`. The network was trained for 8 epochs, with network performance over the epochs shown in Fig. 6.

The final accuracy of the network was 99.1%. This network (actually the network's set of weights) is stored in the file `model_ML_v13_99.1_5conv_512flat.hdf5`.

Deployment. The network runs using Tensorflow 1.13 in Python 3.7. It was tested in Python 3.8 and did not work, so you really need 3.7. The network's operation is directed via `classify_grams.py`, which loads the network weights from the `.hdf5` file, loads spectrograms to be processed, and calls the TensorFlow function `model.predict` to apply the network to the data samples. See below about the structure of FindPorpoises to see how classify_grams is invoked.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 139, 497, 16)      160
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 69, 248, 16)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 67, 246, 32)       4640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 33, 123, 32)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 31, 121, 64)       18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 15, 60, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 13, 58, 64)        36928
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 6, 29, 64)         0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 4, 27, 64)         36928
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 2, 13, 64)         0
_________________________________________________________________
flatten (Flatten)            (None, 1664)              0
_________________________________________________________________
dense (Dense)                (None, 512)               852480
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 513
=================================================================
Total params: 950,145
Trainable params: 950,145
Non-trainable params: 0
_________________________________________________________________
```

Figure 5. Architecture of the network (model) used as the classifier.

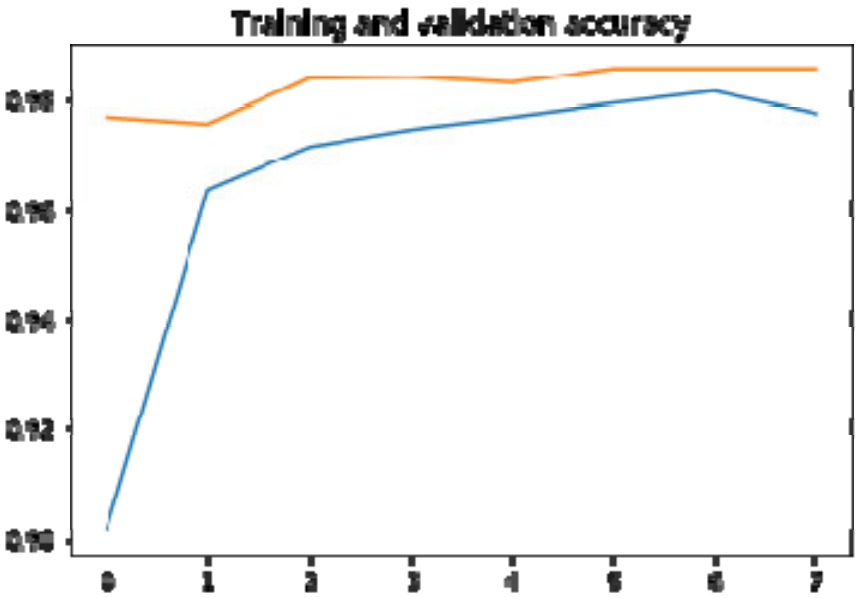


Figure 6. Performance curve during training for the network used in classification.

See Appendix A for the parameter values that control classifier operation and Appendix C for details of the code implementing the classifier.

4. Sequencer

Following the classifier, a sequence analyzer is applied. This relies on the summed autocorrelation function, which takes a time series – in this case, the output of the classifier represented as a spike train – as input and produces as output another time series when there are regularly repeating peaks in the input function. It is windowed, so that it looks at a small (0.5-second) portion of the classifier output at a time, and it is configured to detect regular peaks that repeat with a certain range of repetition rates. See Appendix A for the parameter values that control the sequencer's operation.

5. FindPorpoises

The above steps – detection, classification, and sequencing – are bundled together into a system called *FindPorpoises*. This was compiled using the MATLAB Compiler into an installable Windows application, with the installer called `FindPorpoises1.0_Installer_web.exe`. Included in the installation is a manual, "`Installing and Using FindPorpoises v1.0.docx`", detailing the installation, configuration, and operation of *FindPorpoises*.

6. Display and Tabulation

The process for producing output tables and figures of porpoise occurrence was initially started in MATLAB, using "202003010 Standardized hydroacoustic data report mock-up wRevisedFigures.docx" as

a guideline for the appearance of the final figures. After production of some figures, it was realized that it would be easier in the long run to use R software being developing by Dr. Louise McGarry for displaying similar figures for fish from hydroacoustic data. In part it would be easier because Dr. McGarry's software was nearing completion, and in part because any future changes or additions to the R software would not have to be re-implemented in the MATLAB software. Accordingly, Dr. McGarry sent her code, which was then adapted for use with the harbour porpoise data. Adaptation was necessary because of the different kinds of data used as input: The fish data the code was originally written for had detections of individual fish pingers, while the porpoise data output by *FindPorpoises* represented the minutes throughout each day that harbour porpoises were present.

The porpoise version of the R software, called `plotter_porpoise.R`, takes as input the minute-by-minute tabulation of porpoise presence as output by *FindPorpoises* and produces as output a series of tables and figures that show porpoise presence by week, time of day, tide state, and energy turbine presence/absence.

References

Mellinger, D.K. (2014) Osprey 1.7 User Guide. NOAA Pac. Marine Environ. Lab., avail. from ftp://ftp.pmel.noaa.gov/newport/klinck/FW599/Osprey Documentation - READ ME.pdf .

Mellinger, D.K., S.L. Nieukirk, and S.L. Heimlich. 2018. Ishmael 3.0 User Guide. 77 pp. Avail. from http://www.bioacoustics.us/ishmael.html.

Appendix A: Detection and sequencing parameters

The detection and sequencing steps of *FindPorpoises* are controlled by a number of parameters, shown below.

These are parameters for making spectrograms:

```
'frameSizeS'    1024/512000       % frame size, sec (1024 samples)
'zeroPadFrac'   0                 % zero-padding before FFT
'overlapFrac'   1/2               % frame overlap
'windowFn'      'hanning'         % window type (Hann)
'clipDurS'      0.5               % classifier file duration, sec
'freqRange'     [90000 160000]    % gram is trimmed to this
```

These are parameters for noise-equalization of the spectrogram and energy-ratio detection:

```
'decayTime'     1.0               % exponential-decay time constant
'freq'          [120e3 150e3]     % energy detection band
'ratioFreq'     [ 90e3 120e3]     % energy ratio band
'smoothTimeS'   0.002             % smoothing of resulting detection fn
'threshold1'    0.007             % initial threshold
'threshold2'    0.010             % final threshold (supersedes thresh1)
'durLimitsS'    [0.0 0.5]         % min/max times to be over threshold
```

These are parameters for sequencing of detected clicks:

```
'classThresh'   0.5         % classifier output below this counts as click
'pkSpreadS'     0.01        % for spreading out peaks in click train
'periodS'       [0.025 0.2] % range of acceptable click repetition rates
'windowDurS'    0.5         % window length to run summed autocorrelation
'hopSizeS'      0.2         % how often to run summed autocorrelation
'seqThresh'     20          % threshold for detection after summed autocorr
'mergeTimeS'    5           % successive sequences this close are merged
```

Appendix B: File types

These are the file types that are used as input or produced as intermediate and output files by FindPorpoises:

`*.wav`	sound file; input to FindPorpoises
`*_gram.png`	spectrogram of a sound (.wav) file
`*_ishdet.csv`	detections of candidate clicks made using the energy detection process
`*_class.csv`	output of the classifier (network, model) for each candidate click, with 0 representing 'porpoise click' and 1 representing 'not porpoise click'
`*_seq.csv`	output of the sequencer, with porpoise presence/absence indicated for each minute in the time span analyzed

Appendix C: Structure of the code

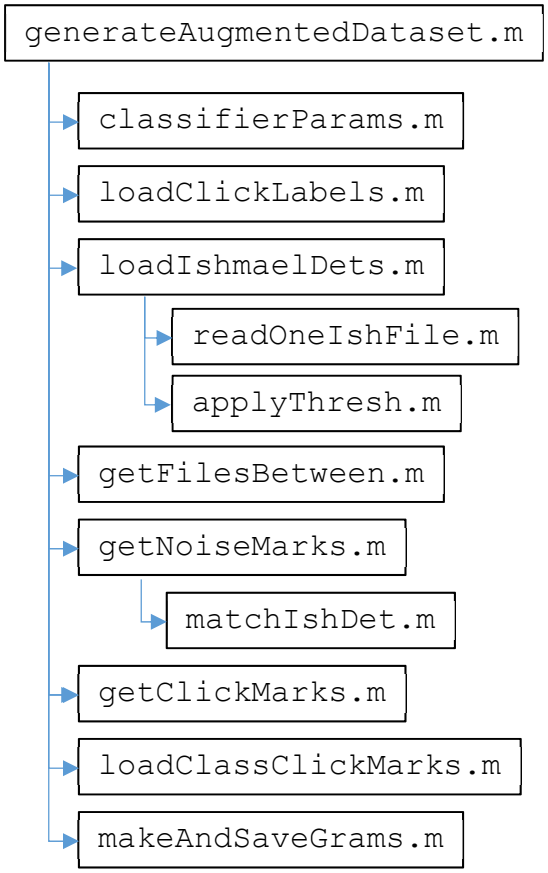
There are three principal phases for creating and using the classifier: preparing the dataset, training the classifier, and running the classifier. Here are the principal modules, and relationships between them, for those phases.

File extensions used in the system:

`.m`	MATLAB code
`.py`	Python code
`.ipynb`	interactive Python notebook
`.bat`	Windows (DOS) batch file
`.r`	R code

Preparing the Dataset

The top module here is `generateAugmentedDataset.m`, a MATLAB script that calls other MATLAB modules as follows:



A key data structure for `generateAugmentedDataset.m` is a 'mark', a structure that represents a ½-second-long segment of sound in one of the sound files. A mark is a MATLAB struct with these fields:

`.fname`	sound file name
`.sam`	sample offset within that file of where the clip starts
`.tSec`	time offset (seconds) corresponding to .sam
`.durSam`	duration of the clip in samples (1/2 s, same for all clips)

If brief, here is what each of the modules above does:

`generateAugmentedDataset.m`: This calls all of the other functions. It has a long configuration section in which are listed all of the sound files that do and don't have clicks, as well as the label files for those that do have clicks. It loads the labels for the labeled clicks, loads the detections found by the detector, and then makes two sets of marks: one set for *Click*, and one set for *NoClick*. It also can load 'class click marks', which are sounds that were erroneously found by the classifier but don't have actual porpoise clicks in them; these provide additional *NoClick* marks. Then the two sets of marks are handed to `makeAndSaveGrams.m`, which uses each mark to create a ½-second spectrogram and save it as a .png file.

`classifierParams.m`: Defines various parameters that go into the classifier. It has three sections, one for spectrogram parameters (`gramParams`), one for detection parameters (`detParams`), and one for sequencer parameters (`seqParams`).

`loadClickLabels.m`: Loads the times of manually labeled clicks.

`loadIshmaelDets.m`: Loads the times when the detector (originally the Ishmael detector, but now done in MATLAB) found clicks. Most of these are false detections and are weeded out by the classifier. These might be contained in several log files.

`readOneIshFile.m`: Load a single log file that has times of detections.

`applyThresh.m`: When the detector runs, it uses a certain threshold (as defined in classifierParams.m) when deciding what counts as a detection. It's possible to raise this threshold when loading the detection times; raising it eliminates some of the detections, potentially a lot of them.

`getFilesBetween.m`: Given start- and end-times, find all sound files present between these times.

`getNoiseMarks.m`: Given a set of files known not to have and porpoise clicks and the times of detections in those files, generate a set of marks centered on the detection times.

`matchIshDet.m`: Given a time, find the nearest detection.

`getClickMarks.m`: Given a set of click logs, make a set of marks randomly chosen (with replacement) from among them. Each mark encompasses a period of time with some minimum number of porpoise clicks in it.

`loadClassClickMarks.m`: Load times that the classifier registered an incorrect positive classification.

`makeAndSaveGrams.m`: Given a set of marks, make spectrograms of the time periods the marks refer to. As an optional means of augmentation, mix in random amounts of noise from a second set of marks.

Also of note is `testClassifier.m`, which makes a set of spectrograms to further test the classifier beyond the spectrograms in the main data set.

Training the Classifier

The classifier network is trained in the Jupyter Notebook `porp_try9_FINAL.ipynb`. (Jupyter Notebooks are installed with the Anaconda3 system; see "Installing and Using FindPorpoises" for information on doing this.) This notebook has a series of cells that can be executed one-by-one via the Run button at top, with not all cells necessary to execute every time. Note that this notebook refers to the network as the "model"; these terms are synonymous.

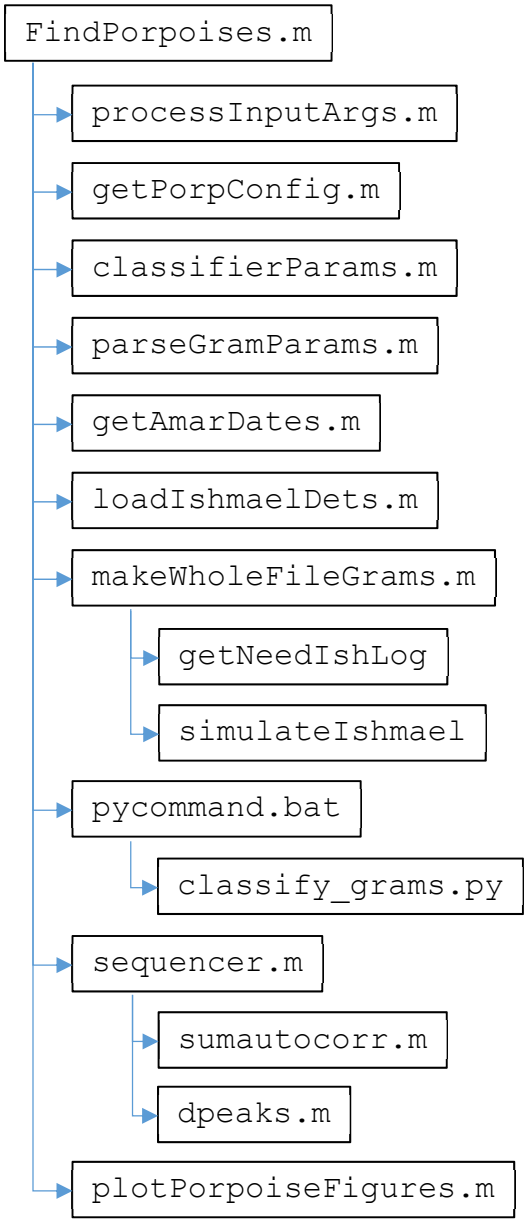
The cells in the notebook are as follows:

- License information cell, nothing executable.
- Cell used as a scratch space for testing bites of code during development; can be ignored.
- Cell to create directory names. Must be run once.
- Cell to prepare for plotting; not necessary to run unless you wish to plot images.
- Cell to plot images; not necessary to run unless you wish to do this.
- TensorFlow installation cell; must be run once EVER on a given computer and then can be ignored. Normally the line that actually does the installation, the one starting "`!pip install tensorflow…`", is commented out so I don't accidentally try to re-install TensorFlow. To run this cell, remove the '#' symbol at the beginning of that line, run the cell, and then (if installation is successful) put the '#' back.
- Cell to define the model. Must be run each time you want to create and train a model.
- Cell to compile the model. Must be run after you define a model in the preceding cell.
- Cell that prepares to get (flow) input data from the data directories. Creates data generators for the training and testing data. Must be run after compiling the model.
- Cell to train the model (model). This is what actually retrieves the training data using the data generators and trains the model (i.e., sets the model weights). This cell can take a while -- several hours on my computer – to run.
- Cell to save the model weights. Run this after training the model if you want to save the resulting model. This cell can save in one of two formats, either as a Pickle file or as a `.hdf5` file, with the one chosen indicated by "`if (True)`" code. I did not have success with Pickle files and now use `.hdf5` exclusively.

- Cell to load a weights file. Use this if you've saved a model and want to re-load it for testing, rather than training a new model.
- Cell to run the model on example cases. This works with `testClassifier.m`, which makes a set of spectrograms to further test the classifier beyond the ones in the main data set.
- Cell to show intermediate representations. This is for seeing how the model operates and debugging it.
- Cell to display accuracy and loss graphically.
- Cell to show the training history. This is mostly a textual representation of what is shown in the previous cell.
- Cleanup cell. This is needed only if you're using Jupyter Notebooks with other notebooks and need to clean up the workspace before using another notebook.

Running the Classifier

FindPorpoises.m runs the classifier. The installable version of FindPorpoises, namely FindPorpoises.exe, is created by the MATLAB compiler, but if you have MATLAB, you can run it as a MATLAB script with more flexibility. It calls other modules as follows:



Here is what these modules do:

`FindPorpoises.m`: This is the main modules that calls all the other ones.

`processInputArgs.m`: Reads the command line, parses flags like "-detect" and "-sequence", and gets the configuration file name.

`getPorpConfig.m`: Reads the configuration file, return a 'cfg' structure with its contents.

`classifierParams.m`: Reads the parameters used by the detector, classifier, and sequencer. This is the same MATLAB routine described above in "Preparing the Dataset".

`parseGramParams.m`: Converts spectrogram parameters expressed in seconds and hertz into numbers of samples and frequency bin numbers.

`getAmarDates.m`: Parses file names to get their date/time information.

`loadIshmaelDets.m`: Loads any existing detection information from `*_ishdet.csv` files for the desired date/time range. This information maybe be absent or partially absent. See `makeWholeFileGrams.m` next.

`makeWholeFileGrams.m`: Makes a spectrogram of each sound file and saves it with the extension "`.gram`". ALSO, if detection information for a given sound file is absent, runs the detection process (energy detector with given frequency bands) using that spectrogram and stores the resulting detections in `*_ishdet.csv` files.

`getNeedIshLog.m`: Determines whether an Ishmael log file (`*_ishdet.csv`, containing detection information) exists for a given sound file.

`simulateIshmael.m`: Runs the energy detection process, applies a detection threshold, finds detections (peaks in the detection function), and stores the times and heights of detections in `*_ishdet.csv` files.

`pycommand.bat`: This is a Windows (DOS) script, called via MATLAB's `dos` function, for running `classify_grams.py` in Python. It is needed because there are certain environment variables that must be defined using Anaconda's "activate base" command. So it executes that command and then starts Python to run `classify_grams.py`.

`classify_grams.py`: This is Python code for running the classifier using pre-trained model weights stored in a `.hdf5` file. It creates a model (network) in TensorFlow, compiles it, loads the model weights from the `.hdf5` file, and loads any detection logs (`*_ishdet.csv` files) present. Then it iterates through the stored whole-file spectrograms (`*_gram.png` files). For each stored whole-file spectrogram (each `*_gram.png` file), it loads the spectrogram and iterates through the detections for that spectrogram; for each one, it extracts a ½-second portion of the spectrogram centered on that detection and calls `predict_arr()` to run the model on it. `predict_arr` in turn calls the TensorFlow routine `model.predict()` to run the model. (Actually it does this in batches of ½-second spectrograms of size `batch_size` for efficiency.) The result of classification is a number typically between 0 and 1, with 0 representing a click classified as a porpoise and 1 a non-porpoise sound. The results of classification for all the detections are written to `*_class.csv` files.

`sequencer.m`: This runs the sequencer. The sequencer loads the `*_class.csv` files with the classifier outputs and applies a summed autocorrelation to them, then detects peaks in the resulting function. The results are then written to `*_seq.csv` output files on a minute-by-minute basis, with porpoise presence/absence indicated for each minute analyzed.

`sumautocorr.m`: This is the summed autocorrelation function, which takes a time series as input and produces as output a function that is high when there are regularly repeating peaks and low otherwise.

`dpeaks.m`: This detects peaks in the function output by `sumautocorr`.

`plotPorpoiseFigures.m`: This is now superseded by the R code for plotting, `plotter_porpoise.R`.