Classification of sounds of *Xenopus laevis*

Henning Thielemann

April 29, 2016

1 Problem

The purpose of our program is to recognise the sounds made by the African clawed frog (*Xenopus laevis*). There are several types of sounds the frog can make, cf. Figure 1, but the most frequent sound is the advertisement call. Technically, the advertisement call is an alternating sequence of rasping and chirping sounds. The program divides recordings into time intervals labelled with the kinds of the sounds and it generates tables for every sound type containing parameters measured for every occurrence of that sound.

2 Solution

If you are only interested in using the program you may skip this section and proceed in Section 3. However, this section gives you insights in the internals and thus should help resolving problems that may occur.

The central part of the classification is a Hidden MARKOV model (HMM)[1]. Its purpose is to make decisions that need a kind of intuition were humans are good at and computers usually fail. The intuition is simulated by a probabilistic model and decisions are made by choosing the most likely explanation for a given recording. A Hidden MARKOV model simulates how a machine switches between certain internal states over time and how it emits observable information depending on the state. Ideally we would define internal states like "advertisement", "rasping", "chirping", "ticking", "growling", "pause" and we would assign every such state with the according noise as observable information. The model can describe sequential dependencies like "average rasping time is 0.6s" and "after rasping, chirping is more probable than pause". Additionally the model connects internal states to observable noise where different states may produce similar noise. The model can resolve such ambiguities by evaluating the sequential dependencies.

We can then utilise the model in several ways:

1. We can apply the model in a forward way: Given the model, we could simulate the frog making sounds.
2. Given the model, an audio recording and a manual classification of the sounds we can judge how likely it is that the audio recording actually contains the claimed sequence of sounds.
3. We can apply the model in a backward way: Given the model and an audio recording we can determine a sequence of states that most likely produced the audio recording. Put differently: The according analysis algorithm can tell a sequence of sounds that we most likely hear in the recording.
4. Given audio recordings and manual classifications we can derive an appropriate model. This is called supervised training.
5. Given a model and audio recordings we can derive a new model that better matches the recordings. This is called unsupervised training.

In principle item 1 would be a funny application but in practice it would not work because the HMM cannot handle audio clips directly. Instead we have to perform some non-reversible processing of audio data, see below for details.

Item 2 might allow the program to judge the reliability of a classification it found. Practically we have not yet checked, whether this works actually. It might be that there is no clear difference between the

[1] `https://en.wikipedia.org/wiki/Hidden_Markov_model`

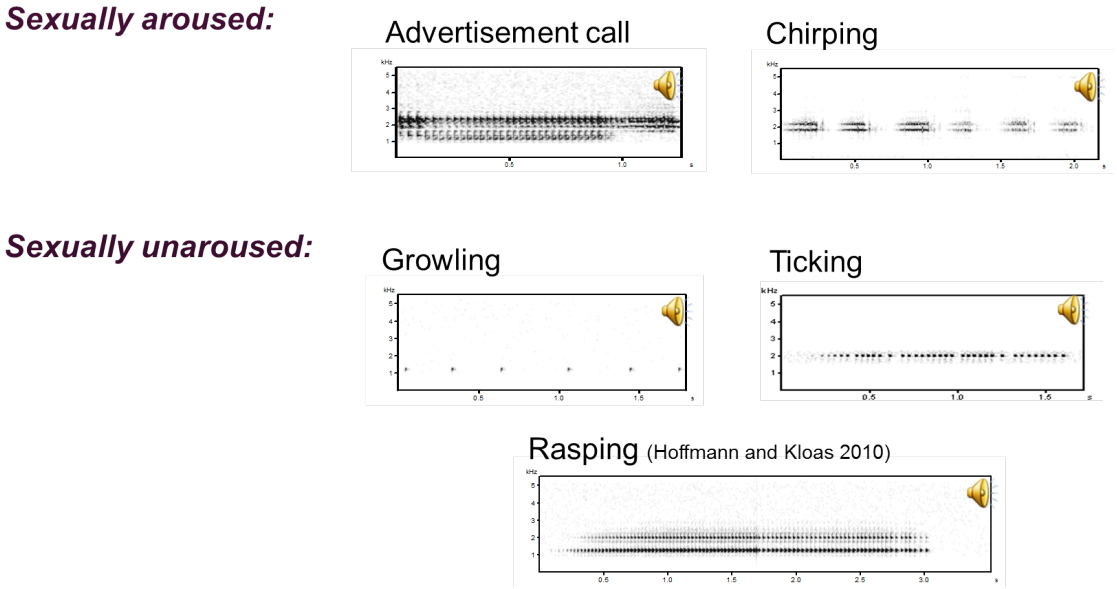

Figure 1: Spectrograms of sound types

likelihoods of correctly and incorrectly classified sounds. However, we can compare the likelihood of two different classifications and can thus judge, which one of two classifications is more plausible.

We use the mode in item 3 to classify sounds within an audio recording.

The supervised training is a pretty natural way of constructing the model. Instead of determining parameters by trial and error, you simply derive the parameters from examples of manually labelled data. In contrast to that, the unsupervised training sounds a bit like magic. How can we improve a model by looking at audio recordings without manual sound classifications? In order to understand that, we must become aware that the somehow dual transformations "supervised training" and "automated classification" both lose information.

That is, say, we construct a model from a manually labelled audio recording. If we let the computer label that recording again using the constructed model we will get a labelling that differs from the manual one. On a second thought this is not surprising: If we mistakenly label two occurrences of a sound once with "rasping" and once with "chirping", then the classification algorithm will have to give them the same name. Consequently, supervised training loses information in general. The model cannot store all information and it is the whole point of its application to abstract from irrelevant details.

Conversely, automated classification is lossy, as well. Say, we have a model that can recognise both rasping and chirping sounds but a recording that only contains rasping sounds but no chirping. If we label the sound using the complete model and then use the labels to train the model again, the information about chirping will be missing. That is, the new model carries less information than the model we started with.

Since both supervised training and automated classification are not invertible, we get constant change when we apply these two steps alternating. In practice, iteratively applying these two steps converges to a stable state where the model reproduces the labels and vice versa. However, the implemented algorithm[2] is even more sophisticated. It does not simply use the most likely labelling of a recording. Instead, it computes a superposition of all possible labellings that are weighted according to their likelihood. Then the model is derived from this weighted superposition.

The bad news are that we cannot directly use the sounds as observable emissions from the model. The classification of sounds using the HMM means to find a sequence of states with maximum likelihood among *all* possible state sequences. This is a rather hard problem that can only be solved if the model is kept simple. Thus, HMMs were designed with efficient computations in mind and we try hard to keep our particular model simple.

[2] `https://en.wikipedia.org/wiki/Baum-Welch_algorithm`

We have to decide at which level we want to run the detection. E.g. the rasping sound consists of a sequence of clicks. We can either try to detect individual clicks with the HMM or we can try to detect the rasping sound as a whole. Chronologically, we tested the first one first. It works in a sense, but the latter one turned out to have much better detection rates. Yet, we will give the details of the first approach since they subsume the parts needed for the second approach.

Figure 2 shows a simple model that we can use for detecting rasping, chirping and a pause. Please note that we must explicitly model pauses. The model cannot stop detecting sounds for parts of the sound, it must always detect *anything*. A pause is not even simple to detect since it contains noise which could be misinterpreted as a weak occurrence of a frog sound. In order to distinguish rasping from chirping sounds we model individual clicks in the rasping sound but the chirping sound as a whole. In particular, we divide each rasping click into the click begin "r0" and the click end "r1". The click begin has a high amplitude and the click end has a low one. The chirping sound is divided into a long body "ch" where the amplitude varies around a middle value and a final phase with decaying amplitude, called "cp".

The numbers at edges in Figure 2 are the probabilities to switch from one state to another one at every time step. They are derived from a real recording.[3] Since it is a single recording the start probability is 1 for "r0" and 0 for all other states. This is not true for models trained with multiple recordings, which should be the normal case. You may verify that the sum of all outgoing probabilities of a state node is always 1. The time step is $5\,\mathrm{ms}$, that is, the probabilities tell us to which state the frog probably switches $5\,\mathrm{ms}$ later. For a better understanding you may watch how rasping clicks are modelled: "r0" has a probability of 0.69 to stay in that state for next 5ms. This means that this phase will be hold for about $\frac{5\,\mathrm{ms}}{1-0.69} \approx 16\,\mathrm{ms}$.[4] After that it will always switch to "r1" and no other state. This way, we ensure that a click always has an end. Or put differently, something that looks like a click beginning with no following click end will not be considered a click, at all. Once in state "r1" there is a certain probability to stay in this state, and if we leave it then most of the times in order to start a new click in state "r0". Less often, but yet possible, a pause or a chirping follows a click. Other transitions from "r1" are excluded. In the training mode the program also forbids certain transitions in order to warn you about mistakes in manual labellings.

The coloured symbols in the state nodes are related to the feature vectors[5] in Figure 4 that we will discuss below.

Figure 3 shows how we prepare data to be analysed by the HMM and how we post-process its results. Data preparation starts with a highpass that removes the humming from the air handler. The following dynamic range compressor amplifies reciprocal to the progression of the volume, which ends up in a signal with constant volume. This way we remove volume variations caused by the movement of the frog. The next step is the actual computation of feature vectors that are suitable for the HMM. We allow the user to choose from a set of feature types in order to experiment with how well they are suited for classification. The figure shows a rather simple, yet effective feature set. It consists essentially of the envelope of two characteristic frequency bands around $1200\,\mathrm{Hz}$ and $2000\,\mathrm{Hz}$. In particular, we apply a bandpass filter that extracts the frequencies along a horizontal line in the spectrogram. We then rectify the signal by computing absolute values, then we compute the maxima over chunks of $5\,\mathrm{ms}$. Thus we end up with a down-sampled signal of $200\,\mathrm{Hz}$ sample rate. We chose this rate as low as possible but such that rasping clicks can still be recognised. The low rate does not only speed up the HMM processing but also improves the recognition. The latter follows from the fact, that the HMM implicitly assumes a geometric distribution for the lengths of runs of a state and thus short runs are more probable then long runs.

Figure 4 shows the feature vectors of the training recording plotted without time information and grouped by the states. For the association of colours and symbols to HMM states, cf. to Figure 2. We see that click begins correspond to high values in both frequency bands and click ends correspond to low values, while the values of both bands are clearly correlated. The effect of the square root is that high values are better concentrated and low values are expanded. This way the sets of feature vectors better match the Gaussian distribution that we use to approximate the feature sets. We also see that the feature vectors for rasping and chirping sounds overlap, but the chirping values are more concentrated. This concentration can be utilised by the HMM to distinguish chirping from rasping. Eventually, the pause feature vectors

[3] `T2014-03-11_18-01-47_0000112.WAV`

[4] `https://en.wikipedia.org/wiki/Geometric_distribution`

[5] `https://en.wikipedia.org/wiki/Feature_vector`

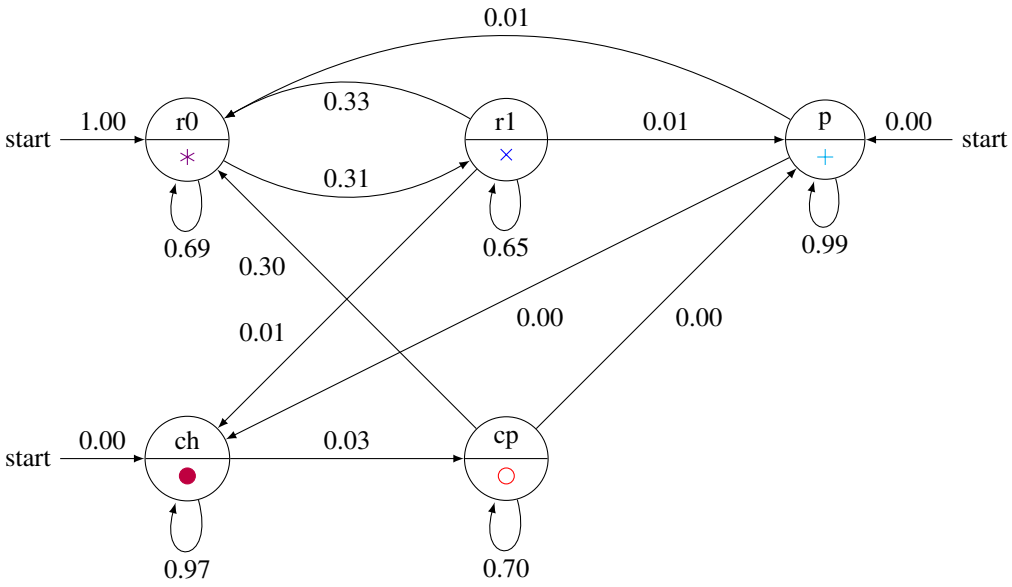

Figure 2: Transition graph derived from real data

can be distinguished from frog sounds since in a pause the energy is equally distributed between the two frequency bands.

Please note that although we use the spectrogram as in Figure 1 as the main representation for manual inspection, we try hard not to use it in the automated processing. This is because the computation of a spectrogram inherently produces artifacts caused by the subdivision into blocks. It depends on parameters like the block size, overlapping and a window where we can hardly justify any particular choice.

In the post-processing we have to apply a small patch. Sometimes clicks are not well expressed and are then misinterpreted as short chirping sounds. We add a post-processing step that converts every short chirping to a rasping click. Subsequently we merge sequences of alternating "r0" and "r1" together to a rasping phase and then merge alternating rasping and chirping sounds to advertisement sounds.

3 Usage

So far, the program can be controlled exclusively from the command line. You get an overview of all functions by running:

```
$ classify-frog --help
```

There are several options and functions that are only useful for experimentation. In this document we only cover the options for production use. In the following sections we will use the slash in file paths as they are used in Linux and Mac systems. In contrast to that, MS Windows uses backslashes in the same places. That is, for MS Windows you must replace e.g. `source/T0123.WAV` by `source\T0123.WAV`.

We have to cope with different file types:

- file name extension `.wav`
  Uncompressed waveform file: We use this format for audio recordings and for signals of feature vectors.
- file name extension `.txt`
  Generally this is a plain text file but in our context it contains a label file for Audacity[6]. You can import it to Audacity with `Import/Labels`. This will add a track with intervals that have text labels.

[6] `https://en.wikipedia.org/wiki/Audacity_(audio_editor)`

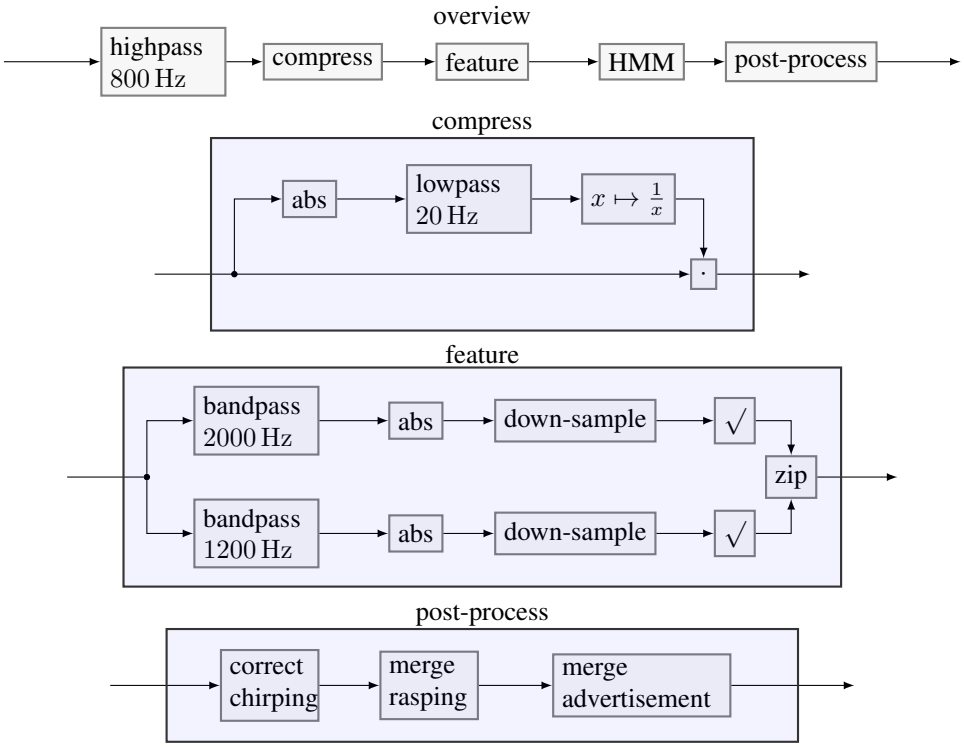

Figure 3: Flowcharts for the classification process

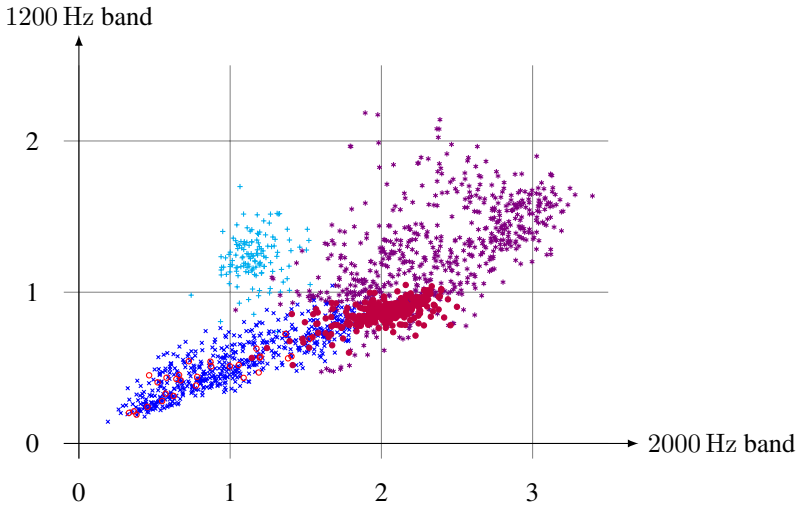

Figure 4: Feature vectors grouped according to the model states. This is computed from the same recording as Figure 2.

- file name extension `.csv`
  This file contains "comma separated values". It is a table in a portable format that can be imported to any spreadsheet software like LibreOffice Calc, OpenOffice Calc, or MS Excel. However, be aware of the problems that are described in Section 4.3. You can enable or disable creation of CSV tables with the options `--emit-csv` and `--no-emit-csv`, respectively. Due to its deficiencies CSV generation is currently disabled by default.
- file name extension `.html`
  This is content written in the "hypertext markup language"[7], i.e. formatted text for display in a web browser. In contrast to CSV it supports highlighted cells, merged cells, hyperlinks and specification of a content language. You can also import such files to LibreOffice Calc and MS Excel, which retains all the formatting. Again, be aware of Section 4.3. You may also import the files into a word processor like LibreOffice Writer or MS Word. You can enable or disable creation of HTML tables with the options `--emit-html` and `--no-emit-html`, respectively.
- file name extension `.xml`
  This is content written in the "extensible markup language"[8] as it is used by Excel 2003. The range of supported features is comparable to that of HTML but it seems to be based on English always, thus it should cause least problems when importing to a spreadsheet processor. You can enable or disable creation of Excel 2003 tables with the options `--emit-xml-2003` and `--no-emit-xml-2003`, respectively.
- file name extension `.aup`
  This is an Audacity project and can be loaded into Audacity with `Open` or by starting Audacity from the command line with

  ```
  $ audacity project.aup
  ```

  See also Section 4.4.

Warning: Like most command-line programs our program overwrites existing files without notice and never asks the user for permission to do it. This is because the program is intended for batch processing.

3.1 Analysis

Analysis is mainly the process of dividing a recording into time intervals and attaching a sound type label to each interval. To this end, the analysis needs an HM model. The analysis process may be complemented with measuring more features per interval.

3.1.1 Single recording

You can analyse a single recording using the following call:

```
$ classify-frog hmm --model=hmm.csv source/T0123.WAV dest/T0123.wav
```

The arguments mean:

- `--model` specifies the model and the name of the feature set to use. The model consists of the probabilities as depicted in Figure 2 written in matrix form. You obtain such a file by a training as described in Section 3.3.
- `source/T0123.WAV` is obviously the name of the input file. It can be stored in any format that SoX[9] supports. E.g. FLAC[10] is a lossless compression codec that may save you about half of the storage space.
- `dest/T0123.wav` is the name of the output file containing the feature signals and it is used to construct the name of other output files, see below. The `dest` directory must exist before starting the classification.

[7] `https://en.wikipedia.org/wiki/HTML`
[8] `https://en.wikipedia.org/wiki/XML`
[9] `https://en.wikipedia.org/wiki/SoX`
[10] `https://en.wikipedia.org/wiki/FLAC`

The program creates the following files in the `dest` directory:

- `T0123.wav`
  This file contains multiple channels, one for every feature. You may need it for investigation when the classification fails. The classification relies entirely on this information. Everything you cannot see in these signals, cannot be seen by the HMM, as well. If everything goes well you will not need that file.
- `T0123-hmm-labels.txt`
  Contains fine-grained labels like "r0" and "ch" as they are detected by the program.
- `T0123-hmm-labels-coarse.txt`
  Here "r0" and "r1" are merged to "rasping" etc. The rasping labels also contain the number of clicks.
- `T0123-hmm-labels-abstract.txt`
  In this file alternating "rasping" and "chirping" labels are merged to "advertisement" calls.
- `T0123-abstimes.txt`
  This file contains intervals showing the absolute times. The times are derived from the name of the input file and the label file is only generated if the input file name contains the absolute start time of the recording.
- `T0123-warnings.txt`
  In this file the programs stores suspicious observations. E.g. it applies two ways of counting clicks in a rasping phase. If they yield different counts it will be reported in an according label.
- `T0123.aup`
  This Audacity project contains the above files as tracks. Loading this project is much easier than importing all files individually.
- `T0123-advertisement-formula.csv`
  This table lists all detected advertisement calls together with several measurements. It contains both measured values as well as values that depend on measurements. In this file we use formulas in order to express dependencies. This helps the user to understand the connection of the values.
- `T0123-advertisement-number.csv`
  This contains the same information as `*formula.csv` but written as plain numbers and not formulas. This is less informative but hopefully more portable. The following files contain the same data for other sound types and follow the same naming scheme.
- `T0123-chirping-formula.csv`
- `T0123-chirping-number.csv`
- `T0123-rasping-formula.csv`
- `T0123-rasping-number.csv`
- `T0123-ticking-formula.csv`
- `T0123-ticking-number.csv`
- `T0123-growling-formula.csv`
- `T0123-growling-number.csv`
- The HTML and XML files contain the same content in other formats.

For a description of the table contents, please see Section 3.1.3.

3.1.2 Multiple recordings

You can analyse a whole multi-night experiment using the following call:

```
$ classify-frog hmmm --model=hmm.csv meto classified +RTS -N3
```

The arguments mean:

- `--model` specifies the model to use.
- `meto` is the name of the input directory. It must contain one directory for every night of the experiment and every night must contain one directory per animal. I.e., if you pass the path `meto`, then the path to a single recording file must look like `meto/Night3/Meto10-8M-4/T0123.WAV`. This fixed structure is necessary since it is reflected by columns of the created tables.
  The program also tries to determine the recording time in order to label classified sounds accordingly and produce per-hour statistics. It determines the recording time either by fetching it from

an Avisoft[11] recording protocol like `meto/Night3/Meto10-8M-4/Meto10-8M-4.LOG` or from the file name itself, if it has a form like `T2014-03-11_23-16-58_0001200.WAV`. Additionally, if a recording protocol is found, then the program checks consistency between the files listed in the protocol and the ones found in the directory.

- `classified` is the name of the target directory. It will be created if it is missing and it will be filled with all results of the computation, see below.
- `+RTS`: This introduces some low-level options that are not handled by our program but the so called runtime environment of the used program compiler. In the future we can hopefully avoid these options.
- `-N3`: Allows the program to utilise up to 3 computing cores. If the machine provides more cores then increasing that number might increase performance. But it may well be that the file handling is the limiting factor and not the audio analysis. You must be careful not to overload your machine since otherwise it may become too busy to react on user input. Thus, try increasing the number step by step until you reach the number of computing cores of your machine. Then watch, whether your machine stays responsive and whether you still get a speedup.

The program puts the following objects in the `classified` directory:

- The directory will be filled with the same sub-directories as the source directory. For every recording the same files are created as in single recording analysis (Section 3.1.1).
- For every night and animal the program generates a summary Audacity project, like `classified/Night3/Meto10-8M-4.aup`. This project contains all recordings of an animal in a night concatenated in a sequence. You should use this for inspection of the results. This saves you from inspection of every single recording. You can even alter labels here and then run the stand-alone measurement as described in Section 3.2.
- Each row in the file `classified/advertisement-recording.csv` contains the five quartiles of all measured values for advertisement calls of a certain recording.
- Each row in the file `classified/advertisement-animal.csv` contains the five quartiles of all measured values for advertisement calls of a certain animal in a certain night.
- Each row in the file `classified/advertisement-treatment.csv` contains the five quartiles of all measured values for advertisement calls of all animals with the same treatment in a certain night. Please note, that the median of medians is different from the median of flattened data, that is, we cannot derive the content of a file with coarse resolution from the data of the files with finer resolution.
- There are analogous quartile tables for all other sound types.
- There are files like `classified/night1/advertisement-hourly-`*treatment*`.ssv` for every night. They contain the portion of advertisement calls for every hour for each treatment.
- The files of kind `classified/advertisement-hourly-`*treatment*`.ssv` contain the concatenation of all hourly files of all nights. These files are intended for plotting. The SSV format ("space-separated values") is especially tailored to gnuplot.
- Finally, the program creates a file called `classified/duration-number.csv` containing the total duration of all recognised sounds per night and animal grouped by sound type. This allows for comparison with manually classified data.

The order of processing the sub-directories determines the order of rows in the created tables. However, the lexicographic ordering that is common for sorting file names can be counter-intuitive for directory names in experiments. You may adapt the order to your needs by inserting one of the following options after `hmmm`:

- `--lexicographic-order`
  This is the standard ordering for files in many programs. It orders this way: `10`, `2`, `3`. In order to get the right ordering, you have to pad with zeros. E.g. we get the wanted order with the names `02`, `03`, `10`.
- `--numeric-order`
  This ordering divides file names in alphabetical and numeric parts. It splits the names at white spaces and dashes, but otherwise ignores these characters. Alphabetical parts are sorted lexicographically

[11] `http://avisoft.com/`

whereas numeric parts are sorted numerically. E.g. we get the order `a5`, `a10`, `b02`, `b11`. This is the most intuitive sorting mode and it is used as default in our program.

- `--custom-order=orderfile.txt`
  You may also explicitly define the ordering of animals by writing their directory names in a plain text file, here `orderfile.txt`. However, you cannot alter the ordering of nights.

3.1.3 Content of spreadsheet tables

The analysis stage creates tables for every recording and call type. Let us examine the data contained in these tables. There are some columns that are provided for all sound types:

- `Source` contains the path of the original recording.
- `Start/s`, `Stop/s` are beginning and end times of the call within the original recording.
- `Duration/s` is the duration of the call.

For every call or call part there is a set of features that is common to all sound types. We call them the spectral features and describe them at the end. It follows the list of features that are specific for every call type:

Advertisement calls

- `SlowDur/s`, `FastDur/s`: These are the durations of the slow and the fast part of the advertisement call. The columns after these duration columns are equal to the features of the rasping and the chirping sound.

Rasping

- `NumClicks` is the number of clicks in a rasping sound.
- `NumEmphasized` denotes the number of initial emphasised clicks. These clicks are noticeably louder than the following clicks. Unfortunately the amplitude of the sound may vary for other reasons, e.g. the frog moving away from the microphone. Consequently we have to resort to a heuristics: We assume that without emphasised clicks the envelope of the click sequence would be a linear one. We use linear regression to determine that envelope. Additionally we assume that there are only a few emphasised clicks that do not much distort the linear regression. Finally, we select all initial clicks until they fall below 10% above the linear envelope.
- `ClickRate/Hz` is the average number of clicks per time.
- `ClickHalfLife`: The half-life is the duration from the beginning of a click until its envelope reaches half of the initial amplitude. This column shows the average of the half-lifes of all clicks in a rasping sound.
- `ClickPause`: The click pause is the duration from the half-life to the beginning of the next click. This column contains the average for all clicks.

Chirping

- `MainDur/s`: Every chirping sound ends with a release phase. `MainDur` denotes the duration of the sound without the release phase.

Ticking

- `NumClicks`, `ClickRate/Hz`: These are the same as in rasping calls. However, the program cannot count the clicks very reliably. Furthermore the clicks might be distributed irregularly, in which case a click rate does not make much sense.

The program also computes a set of features that describes the distribution of frequencies. These features are computed equally for all call types.

- `WienerEntropy`: This is the WIENER entropy or spectral flatness[12]. It is computed on blocks of size 256 with overlap factor 2 and averaged over all blocks.

[12] `https://en.wikipedia.org/wiki/Wiener_entropy`

- `MaxFreq`: The frequency with maximum amplitude is determined for every spectrum block and averaged over all blocks.

Features based on the block FOURIER transform depend on quite arbitrarily chosen parameters like the block size, the overlap factor and the block window. Thus we also provide the following non-FOURIER features.

- `BandCentroid/Hz`, `BandDeviation/Hz`: These features are computed from the volumes of the frequency bands at 1 kHz, 2.5 kHz, and 4 kHz. These are roughly the dominating frequencies of the frog calls. Due to the taken computing approach the centroid frequency is always between 1 kHz and 4 kHz.
- `SpecCentroid/Hz`, `SpecSpread/Hz`: The spectral centroid and spectral spread are strictly spoken features computed from the FOURIER transform.[13] However, we use approximations involving the volume of the first and the second derivative of the signal, that do not need a FOURIER transform at all.

3.2 Measurement

Analysis of a multi-night experiment (Section 3.1.2) always includes computation of features of the determined sounds. However, if you later find that some sounds were misclassified then the computed features become worthless. That is why we provide the computation of features as a separate step.

You may run the analysis as described in Section 3.1.2. Afterwards you can inspect one of the Audacity projects like `classified/Night3/Meto10-8M-4.aup` that summarises all recordings of a night and an animal. You may modify the "classes" label track and only this one. All other tracks must be left untouched. You must also take care that labels in the "classes" track do not cross boundaries of recordings. The numbers in the "rasping" and "advertisement" labels represent the number of clicks found in the calls. They are only for your information and you may leave or delete them. Save the project under the existing name (i.e. `File/Save Project`, Ctrl-S).

Once you finished correction of the classification you run the feature computation with this command:

```
$ classify-frog measurem classified measured +RTS -N3
```

This will read all summary Audacity projects and examine the referred original recordings according to the (modified) "classes" tracks. The click numbers shown in "rasping" and "advertisement" labels are ignored. The structure of the `measured` directory is almost the same as the one of `classified`. One notable difference is that warning label tracks are no longer generated. From a user point of view this makes sense since the program expects that the user has resolved all conflicts. From a technical point of view this makes sense since many warnings report inconsistencies between the classification and the feature computation phase, and in the measurement operation the first one is missing.

Section 3.4 shows how to use the measurement in connection with classification.

3.3 Training

The goal of a training is to obtain a file that we can pass to the `--model` option of one of the analysis operations described in Section 3.1. Using training we can both create new models and adapt old models to new data. The first kind of training is called supervised training and the second kind is called unsupervised training.

3.3.1 Supervised training on a single recording with fine-grained labels

```
$ classify-frog trains source/T0123.WAV dest/T0123.wav
```

This command expects an audio recording in `source/T0123.WAV` and a fine grained labelling containing labels like "r0" and "r1" in `source/T0123.txt`.

[13] `https://en.wikipedia.org/wiki/Spectral_centroid`

You must make sure that every point in time is covered by exactly one label. The program tolerates small gaps or overlaps, but you may not leave large gaps. A pause should be marked with a "p" label. We could alter the program and make it interpret all large gaps as "p" but so far we think it is an additional safety belt not to do so.

You may add more options after `trains`:

- `--feature=band-1200hz-2000hz-low-rate-200hz-sqrt` lets you choose a particular feature set (see Figure 3). The `band-1200hz...` set is the default, which we also used to produce Figure 4. You get a full list of all available feature sets when you run the program with the `--help` option.
- `--plot` lets the program plot all feature vectors (or better: points) grouped by labels, i.e. by states. This helps to judge whether the feature set is appropriate for the analysed data. A suitable feature set should produce separated clouds of points for every label type. It requires that the plotting program gnuplot[14] is installed.

The program generates the following files in the `dest` directory.

- `T0123.wav`
  This file contains the features in its channels.
- `T0123-hmm-supervised.csv`
  This is the model computed from the given labels using supervised training. You can use it as an argument to the `--model` option. You may also load it into a spreadsheet application for inspection.
- `T0123-hmm-unsupervised.csv`
  After the supervised training the program automatically repeats an unsupervised training until convergence and writes the limit model to the above file. You may prefer this model to the first one since minor flaws in manual labelling might have been corrected. But it can also be that the unsupervised model is completely unusable. After the unsupervised training the program emits the maximum difference between probabilities of both transition graphs. If the difference is small, i.e. less than 0.1, then everything went well. A large difference means that unsupervised training has turned the model into something completely different.
- `T0123-hmm-labels-supervised.txt`
  After supervised training the program starts analysis of the recording using the freshly constructed model. It stores the result in the above label file. You can watch those labels in order to verify what information is really captured in the model after the training.
- `T0123-hmm-labels-unsupervised.txt`
  This is the same for the unsupervised training.
- `T0123.aup`
  The Audacity project file contains the above tracks in one handy file.

3.3.2 Supervised training on a single recording with coarse labels

```
$ classify-frog trainsc source/T0123.WAV dest/T0123.wav
```

Setting fine-grained labels for the previous command is very laborious and it must be done for a specific feature set. This command works like the above one but expects a file named `source/T0123-coarse.txt` containing labels like "rasping" and "chirping". In the first step it tries to automatically divide the coarse labels into fine ones and stores them in `dest/T0123-fine.txt`. This will be used for the actual supervised training. Unfortunately, the program can already make errors on refinement. You can check that by watching the contents of `dest/T0123-fine.txt`. Additionally you can hope that the subsequent unsupervised training will eliminate such errors.

3.3.3 Supervised and unsupervised training with multiple recordings

```
$ classify-frog trainm source/* --output trained
```

[14] `https://en.wikipedia.org/wiki/gnuplot`

This runs a supervised training on a set of recordings where you can mix finely and coarsely grained label tracks. After the supervised training an unsupervised training is repeated until convergence. For every recording a set of output files is written to the `trained` directory. The set of output files is the same as in the training of single recordings.

You can specify a mixed list of audio and label files. It is most easiest to achieve using file name patterns[15] such as `source/*`. The program automatically associates the files according to their filename extensions. Recording files with corresponding label files are fed to the supervised training. Subsequently, all recording files, that is, both labelled and unlabelled ones, are used for unsupervised training.

The program emits two results, namely the model `hmm-supervised.csv` as found by the supervised training and the model `hmm-unsupervised.csv` as found by the subsequent unsupervised training.

3.3.4 Unsupervised training with multiple recordings

```
$ classify-frog trainum --model=hmm.csv source --output trained
```

If you have a model `hmm.csv` that worked well for a set of recordings, you may encounter that it does not work well on a different set. In this case you would have to manually create new label tracks. However, unsupervised training might come to a rescue. If you run an unsupervised training with the old model `hmm.csv` and new sound files in the directory `source` then you may have luck and it produces a new model which fits to the new data. Again, the program shows the difference between old and new model and a low value indicates that everything went well.

Be aware, that the unsupervised training does not have the recordings at hand that generated `hmm.csv`. The old model might not even be created by training. That is, unsupervised training can only adapt to the data you give it here. Conclusively, if a certain sound is not present in the new data it will be lost in the model. In this case some transition probabilities in the model will drop to zero and it is likely that you can detect this case by a large difference between old and new model.

3.4 Best practice

Eventually we want to give a workflow that should enable the most efficient use of our program. Start with analysing a bunch of data that is stored in a directory, say `meto`, as in Section 3.1.2:

```
$ classify-frog hmmm --model=hmm.csv meto classified +RTS -N3
```

While the program runs you can already start to inspect the generated CSV and AUP files. We advise to watch the summary Audacity projects that are generated per night and animal, like `classified/Night3/Meto10-8M-4.aup`. This allows you to quickly verify a lot of recordings and, as a last resort, it would be also the basis for manual classification. You may manually scroll through the sequence of recordings while checking correct classification, but you may also use the tabulator key in the "warnings" label track to quickly jump to intervals that the program itself found suspicious. E.g. if counting clicks by two different methods yields different click numbers then this is reported as a warning. The "warnings" label track is also stored as separate file. You can use this to select recordings with probably many misclassified sounds. Just sort the files in a graphical file manager with respect to their size. You may also run

```
$ ls -lS classified/Night3/MetoKo1/*warnings.txt
```

in a Linux terminal.

If you find that too many sounds are misclassified then you may consider updating the Hidden MARKOV model. You can do this using the unsupervised training as described in Section 3.3.4. Unsupervised training allows you to adapt an old model to new data. In order to run it, you must carefully select a set of recordings for training:

[15] `https://en.wikipedia.org/wiki/Wildcard_character`

- The set must contain at least one occurrence of every call type. If a call type is missing then the program will not be able to detect this call type with the generated model, anymore. You may fulfil this criterion by always adding a set of standard recordings for every call type.
- Even more, every possible pair of consecutive sounds and every possible sound at the start of a recording must occur at least once. If this is not the case the program will assume that certain sounds are not allowed as start sounds or are not allowed to follow certain other sounds. The program will then circumvent this by inserting short phases of unrelated sounds such that only trained sound transitions occur. This condition seems to be overly strict but the HM model stores the frequency of every transition in order to find most plausible transitions in an analysis run. If a training set does not contain a certain transition then its observed frequency is zero.
- The training set should contain not only recordings where classification fails with the old model. It should also contain recordings where classification went well, so far.
- The relative number of recordings counts. That is, you may duplicate a recording under different names and get different training results. This happens since training adapts more to data that is present more frequently in the training set. It follows that you should select a representative subset of recordings for training.

We advise to copy all training recordings to a directory like `training`. This simplifies running the training and is useful for documentation of how the new model was generated. Then run the training as follows:

```
$ classify-frog trainum --model=hmm.csv training/*.WAV --output trained
```

If you do not like to copy training recordings, you may replace `training/*.WAV` by a space separated list of training file names.

The main result is `trained/hmm-unsupervised.csv` that you can pass to the `--model` option in future `hmmm` runs. You may copy that file to the directory of your classification output in order to document with which model you analysed the data of an experiment.

After the training the program reports the difference between the old and the new model. If the difference is small, say below 0.1, then everything went well. Additionally you may check how the new model classifies your data. For instance you may run single file analysis with `hmm` on some training recordings, or re-run analysis of the whole experiment as before.

If the difference of models was too high then you may add more recordings to the training set that are well classified with the old model. Alternatively you may remove misclassified recordings from the training set, then run the training and if this works well, re-add problematic recordings to the training set and run the training starting from the intermediate model. This way you can shift the old model successively to new data. If we find that we need this procedure more frequently and it proves useful then the program could be even extended to perform this procedure automatically.

If nothing of the above helps then we are in serious trouble. I will show further ways to successfully complete a training but I doubt that it will help much. You may fall back to supervised training, that means you need recordings where the different calls are already labelled. Consider these hints:

- The supervised training with multiple data sets is always complemented with an unsupervised training. That is, you may label only a part of all training recordings. See Section 3.3.3 for details.
- All call types and all possible transitions between call types must be present in the labelled portion of the recordings set. If a call type is missing, it will be missing in the model and the program will not be able to detect this call type anymore. If a transition is missing, the program will insert short phases of unrelated calls when it encounters this transition in recordings.
- You have the choice between fine and coarse grained labels, see Section 3.3.1 and Section 3.3.2. First try coarse grained labels, since there is less to label, they are more intuitive and can be re-used for other feature sets. The program refines the coarse labels for you. Check in Audacity whether the refined labels are acceptable, and if they contain errors, check whether they are still present after labelling with the unsupervised model. Only if all this fails, too, fix the fine-grained labels produced by the program manually in Audacity. Keep in mind that the fine-grained labelling depends on the feature set. Other feature sets might use the same label names for slightly different purposes or may use completely different sets of label names.

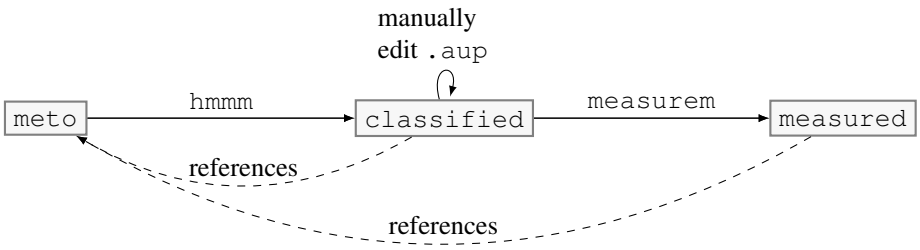

Figure 5: Flowchart for classification with manual corrections

- You should not even mark the intervals manually but instead copy "coarse" label tracks from an analysis run. It might be faster to correct misclassified label tracks in Audacity instead of creating new ones.

If you have prepared the labelled training set then you can run the training:

```
$ classify-frog trainm source/* --output trained
```

If the generated model does not work well you may still try different feature sets, by adding the `--feature` option after the `trainm` option. We do not expect that it helps much, though. Btw. it is not possible to change the feature set via an unsupervised training.

If all the training efforts fail then you must manually classify your data as a last resort. Yet, our program can assist you! Even for manual classification you would run the batched analysis operation as above. Then load summary Audacity projects into Audacity and adapt the "class" tracks or replace them by new ones. Then run the measurement operation as described in Section 3.2 and the program will compute all features of the classified calls:

```
$ classify-frog measurem classified measured +RTS -N3
```

Figure 5 shows how you should first classify your data, then correct it manually using Audacity and finally process the manually corrected classification in order to get tidied spreadsheet tables. It also shows how the Audacity projects still contain references (paths) to the original recordings. This shall remind you that you should not move the original recordings away (see Section 4.4).

4 Trouble shooting

4.1 Error messages

There are certain situations where the program aborts with an error message. The error messages should be self-explanatory in most cases but sometimes there are non-obvious causes.

- `empty intervals found in T0123:`
  `4.065:  r1`
  This means that there is an interval at 4.065 seconds in the recording `T0123` of zero length. This error may even happen if the interval is longer than zero seconds, but still quite small. The problem is that the program uses a certain resolution for analysis that is below the sample period. The resolution depends on the feature set but is most often 5 ms (200 Hz). At this low resolution the label interval might collapse to zero. To resolve the problem you have to make the interval larger than 5 ms.
- `detected forbidden transitions in T0123:`
  `p -> r1`
  As an additional safety belt every feature set has a set of admissible transitions. In theory, forbidden transitions can occur in valid label tracks if too short intervals are neglected, but this should be catched by the error above.

- `training/T0123.wav: openFile: does not exist (No such file or directory)`
  This may be confusing when `training/T0123.wav` is the name of an output file. It should not be an error if it is missing! However, the message may mean that actually the directory `training` is missing.
- `Meto/Night1/Meto-10-10M-1/Meto-10-10M-1.LOG: getDirectoryContents: invalid argument`
  The message means that the program tried to fetch the contents of the directory `Meto-10-10M-1.LOG`, which failed because `Meto-10-10M-1.LOG` is just not a directory but a file. The likely cause is that you did not pass the right directory to the program. In this case it must be `Meto`, not `Meto/Night1`.

4.2 Command-line

On MS Windows the program will fail loading audio recordings from paths containing umlauts and other special characters. This is because the recordings are loaded using `libsox` and it is not documented how it copes with special characters. We will hopefully solve the problem, but be warned for now.

On MS Windows' command line you must not add a trailing backslash to a directory argument. E.g. write `meto` not `meto\`. In contrast to that, on Linux the path `meto/` would be tolerated and the trailing slash is automatically appended by tabulator key completion (see Section 5.1).

4.3 Import of table files to a graphical spreadsheet processor

Our program exports its results in tables, formatted in several text-based formats like CSV, HTML and XML. These formats are easy to generate and reasonably portable between different spreadsheet processors like LibreOffice Calc and MS Excel. However, the interpretation of the content is not well standardised and even inconsistent within one program. You may load such a file into a plain text editor in order to see its raw content. This way you can verify whether our program emitted wrong results or whether the import to the spreadsheet processor failed.

Here are some known issues:

Locales: For CSV files our program uses commas as table field separators and decimal points for numbers. If you import that to a spreadsheet processor on a machine configured for German users then the decimal point is misinterpreted as a thousands group separator.

- **LibreOffice Calc:** If you open a CSV file then LibreOffice will show you the CSV import dialog. There you must switch to an English interpretation of the CSV file content. However, even then numbers and function names in formulas are interpreted the German way if you have configured LibreOffice for German language. E.g. LibreOffice expects "SUMME" in formulas instead of "SUM". Thus you may be better off configuring LibreOffice for English language or ignore the formula based CSV files. We reduce dependency on locales in formulas by using ratios instead of decimal fractions.
- **MS Excel:** If you load a CSV file into MS Excel then MS Excel will store all the content in the first column of the spread sheet. Afterwards, you can convert it to a table using the "Text-to-Columns" operation. To this end, mark the column containing the text, then select `Data/Text-to-Columns` and choose the following items in the dialogs:
  - Step 1 of 3: "Delimited"
  - Step 2 of 3: Delimiters: `,`, Text qualifier: `"`
  - Step 3 of 3: Decimal separator: `.`, Thousands separator: ` `
    It is important not to enter the characters via keyboard, but to select them from the drop-down-menu. Otherwise your selection will be ignored.

HTML format allows for more features than CSV but the problems with importing numbers and formulas seem to be even worse. Both LibreOffice and MS Excel ignore the content language specified in an HTML file. In LibreOffice you have to choose "English" manually when importing. In MS Excel there

seems to be no way to switch to an "English" interpretation. That is, English numbers and formulas may not be imported properly to MS Excel.

The Excel 2003 XML format should cause the least problems with both numbers and formulas.

Locking: If you open a CSV file with LibreOffice on MS Windows it will be locked such that our program fails to write to it and eventually aborts. Instead you may load the plain text to NotePad. NotePad does not lock the file and instead offers you to reload the table if it is extended by new rows. If you want to see updates of a CSV file instantly then you may install the program WinTail. The downside of both programs is that they show plain text rather than tables. If you want to see formatted data, then better use a browser to watch the generated HTML files. During construction the HTML files are incomplete and thus not valid, but most browsers accept them anyway.

4.4 Audacity

An Audacity project named `recording` always consists of two things:

1. The file `recording.aup` contains the layout of all tracks, including track names and label track contents. It is a human-readable text file in XML[16] format.
2. The directory `recording_data` contains binary data, e.g. waveform overviews and sometimes original waveform data.

When moving or renaming an Audacity project you should keep in mind that both parts must be moved or renamed. You need even more care when moving the original recordings. Our program does not embed original recordings in its generated Audacity projects since this would duplicate space usage. Instead it stores the paths of the original data in `recording.aup`. When moving the original data you must update its paths in `recording.aup` using search&replace in a text editor.

5 Tips and tricks

5.1 Command-line

Entering paths: Typing lengthy directory and file paths into the command-line window is cumbersome and error-prone. There are two ways to simplify that work equally on Linux and Windows:

1. Type the beginning of a path and press the tabulator key. If the entered prefix uniquely determines the path then the command-line processor inserts it for you. If it is ambiguous then the reaction depends on your system. On Linux a short acoustic or visual warning is emitted and pressing the tabulator key again, lists all objects with the given prefix. On MS Windows the path is completed to the lexicographically first matching path. Additional tabulator key hits cycle through all matching paths.
2. Drag an icon in a graphical file manager with the mouse and drop it into the command-line window. The position of dropping is ignored, the path is always inserted at the position of the command-line cursor.

Batch files: It is a good idea to write working command line calls to text files. On Linux they should get the file name extension `.sh` (read: "Shell script"). On MS Windows the file name extension should be `.bat` (read: "Batch file"). Double clicking on a batch file[17] in the Windows Explorer runs the commands in the text file line by line. Windows will open a command window where all text output and input from and to the programs will go. Unfortunately, the window will be closed immediately after the termination of the programs. This way, error messages might get lost. You can keep the command window open in two ways:

- Add a `pause` command at the end of a batch file.

[16] `https://en.wikipedia.org/wiki/XML`
[17] `https://en.wikipedia.org/wiki/Batchfile`

- Open a command line window manually (program name "cmd") and run the script there. Remember the tricks in Section 5.1 on how to enter the location of the script.

5.2 Audacity

Spectrograms: Audacity provides a nice interactive spectrogram view of audio data, but it is not obvious how to enable it. This view must be set per track. In the left upper corner of each track there is a box containing the track name with a triangle pointing downwards on the right hand side. If you click on this box you get a pull-down menu where you can switch from "Waveform" to "Spectrogram". You can modify the spectrogram parameters in `Edit/Preferences` (Ctrl-P).

Unfortunately, Audacity-2.1 does not store the kind of view in its project files. Thus the frog sound classification program cannot enable the spectrogram view by default. You can check whether your version of Audacity stores the view type by saving a project with a spectrogram track to an AUP file and load it back to Audacity, again.

Label tracks: You can jump to the next and the previous label with TAB and with Shift+TAB, respectively. You can add a new label by selecting an interval and then choose `Tracks/Add Label At Selection` (Ctrl-B). Labels are added to the label track below the waveform track. If there are multiple label tracks you may have to re-arrange them. You can export label tracks but not individually. If there are multiple label tracks and you select `File/Export Labels ...` then all label tracks are merged into one track and written to a file. This is certainly not what you want. You may still load such a file into a text editor or spreadsheet application and remove the unwanted labels. If you use labels to manually classify recordings then it is best to use only one label track.