Parrot integration

Document version 1.0

What is the Parrot integration?

A customizable API which allows you to use more noises than pop and hiss. Noises are sounds that are recognized 60 times a second, allowing for a near real time response.

Requirements

1. Access to the TalonVoice Beta - Parrot integration is an experimental API which requires you to have access to the beta build of Talon.

2. The parrot_integration.py file which hooks up everything together. This file should be placed in the talon/user directory.

3. A Parrot model: This is the neural network model that recognizes sounds based on the audio input. You can train your own personalized model in Parrot.PY or use a pretrained model.

A readily trained model that I trained can be found in the models directory on https://github.com/chaosparrot/parrotpy_tryout_bundle/tree/master/models .

Use the parrot_ensemble_12.pkl and place it in your parrot folder in your talon home directory as model.pkl .If the parrot folder does not exist yet, create it in your talon home directory (one folder up from the user directory)

4. patterns.json - This file allows you to define your own noises and when they get recognized. For example, you can define a loud whistle, a soft whistle, a low or a high pitched whistle all on your own without having to change your model.

This file should also be placed in your parrot folder in the talon home directory
An example to have a tongue clicking sound and a shushing sound working with the model above would be:

parrot/patterns.json

```
{
    "cluck": {
        "sounds": ["click_alveolar"],
            "threshold": {
            ">power": 20,
            ">probability": 0.9
        }
    },
    "shush": {
        "sounds": ["sibilant_sh", "sibilant_ch"],
        "threshold": {
            ">power": 20,
            ">probability": 0.9
        }
    }
}
```

5. Talon files where you can respond to recognized noises given the files context
An example to get cluck to click working would be:

example.talon

```
parrot(cluck):
    mouse_click(0)
```

So in short, this is how the integration works

- When the noise mode is active, sound from the microphone gets processed by the Parrot model (model.pkl)
- The result gets put against the pattern matching (parrot_integration.py), which determines the noises based on the defined patterns (patterns.json)
- The names of the noises are then used by Talon to determine what commands to execute from the .talon files

The .talon file syntax

```
parrot(shush): print("Activates at the start of a shushing sound")          .talon
parrot(shush:repeat): print("Activates during the shushing sound")
parrot(shush:stop): print("Activates when the shushing sound is stopped")
```

As the noise commands are placed in .talon files, you get access to all the context matching that they to offer. You can make the cluck noise do something else entirely in a different program, or in a different mode.

In the example below, the cluck sound can be made in dictation mode to delete a word, instead of left clicking like in the example in the requirements.

```
mode: dictation                                   example_clear_word.talon
-
parrot(cluck):
    edit.extend_word_left()
    key(backspace)
```

For more advanced usecases like scrolling based on how loud your sound is, you can make use of some values in parrot noise commands to tweak your results.

ts is the timestamp of the noise, when the noise is made
power is a measurement on how loud the audio is
f1 is a measurement of the first formant frequency
f2 is a measurement of the second formant frequency

More on what these measurements represent is given in the in-depth section up ahead

The .talon file below scrolls down when you shush, and when it is louder, it will scroll down faster

```
parrot(shush:repeat):                             example_scroll.talon
    mouse_scroll(power)
```

The patterns.json file syntax

Be aware that the changes to patterns.json might not immediately activate, you will have to restart Talon to make them active. If you are unsure of the proper format, look at the debugging section below, or simply use a tool like https://jsonlint.com to check if it is the right format.

parrot/patterns.json

```
{
    "whistle_high": {
        "sounds": [“sound_whistle”],
        "threshold": {
            ">probability": 0.9,
            ">power": 30,
            ">f1": 600
        }
    },
    "whistle_low": {
        "sounds": ["sound_whistle"],
        "threshold": {
            ">probability": 0.9,
            ">power": 40,
            "<f1": 600
        },
        "detect_after": 0.1,
        "graceperiod": 0.1,
        "grace_threshold": {
            ">probability": 0.6,
            ">power": 10
        },
        "throttle": {
            "whistle_high": 0.5
        }
    }
}
```

The “whistle_low” and “whistle_high” part are the names of your noises which you would use in the .talon files

The **sounds** part determines the Parrot model output labels to use to detect this noise.

detect_after is the amount of seconds the noise will have to be made before the first parrot(): result gets activated. In the example above, “whistle_high” will only activate after 0.1 seconds, or 100 milliseconds of the sound being made

threshold is where we determine when this noise should be detected
In the example above, whistle_high will only trigger when the probability of ‘sound_whistle’ reaches above 90%, when f1 frequency is higher than 600Hz and the power value is above 30.

The thresholds you can use are:

>probability - Probability of the sounds needs to be above this value to activate. This is a value between 1.0 and 0.0, where 1.0 means 100%
>power - Power needs to be above this value to activate
>f1 - F1 needs to be above this value to activate
>f2 - F2 needs to be above this value to activate
>ratio - This can only be used when multiple sound are in a pattern. The value being passed is probability of sound 1 divided by probability of sound 2 to determine the ratio between them.

For all these higher than thresholds, there are lower than equivelants as well
<probability - Probability of the sounds needs to be belowthis value to activate.
<power - Power needs to be belowthis value to activate
<f1 - F1 needs to be below this value
<f2 - F2 needs to be below this value
<ratio - Activate below this ratio

The grace period values can be used to slightly tweak your noise requirements after activation. When you are making a noise for a longer time, the sound changes and that will affect the Parrot model output.
Setting the **graceperiod** value to 0.1 seconds will activate all the thresholds in **grace_threshold** after a noise has activated, so you can make the requirements easier on your voice.
In the example above, the grace period f1 requirement of “whistle_low” is dropped and the power requirement is lower, allowing you to change the pitch and loudness of the sound without the noise being deactivated.

throttle is where you can turn off matching for any noise after the noise has been made.

For instance, the example above disables “whistle_high” for half a second when “whistle_low” is activated.
The format is {noise name}: {amount of seconds to disable} , and multiple noises can be deactivated at the same time

Debugging your noises

To debug your noises and patterns, or to create new patterns, you change debug=-False in the parrot_integration.py file at the bottom to debug=True to show more in depth information in the events.tail()
Make sure that when you are done debugging, that you turn the debug=True back to debug=False, otherwise your logs will be filled with parrot events, which will overwhelm the other debug information.

To look under the hood to see what the model is outputting, open up the Talon repl program, which can be found in bin/repl in your Talon home directory, and type events.tail()

Now you can talk or make sounds, and if debug is turned on, it will output a bunch of predictions that looks something like this:

```
thread<2812> | parrot predict sibilant_s 100.00% pow=65.61 f1=500.210 f2=5834.537
thread<2812> | parrot predict sibilant_s 100.00% pow=65.41 f1=276.290 f2=5209.802
thread<2812> | parrot predict sibilant_s 99.99% pow=76.73 f1=359.931 f2=5558.041
thread<2812> | parrot predict sibilant_s 99.99% pow=82.38 f1=547.690 f2=5536.604
thread<2812> | parrot predict sibilant_s 100.00% pow=78.98 f1=183.160 f2=5415.111
```

If the repl logs are moving too fast, you can click in the content and scroll freely without the logs going out of view by new logs.
Lets say we wanted to add a pattern named hiss and activate it when these sounds are made.
We can see that the lowest power is something like 65, the F1 value seems to jump around a bit, and the F2 one is steady above 5.2kHz.
That means we can add a pattern like this

parrot/patterns.json

```json
{
    "hiss": {
        "sounds": ["sibilant_s"],
        "threshold": {
            ">probability": 0.999,
            ">power": 65,
            ">f2": 5200
        }
    }
}
```

And when we next restart talon, we can use parrot(hiss) in our .talon files for activation.

If you are having trouble getting the patterns to recognize, make sure to look in the logs for [parrot] during start up.
These log messages offer you tips on your patterns.json format, whether the pattern is activated or wheter it is invalid because of some incorrect value.

An example of a warning you might see is this:

```
2021-05-22 12:19:39 WARNING [parrot] pattern ‘shush’ contains invalid labels and will not be used.
```

Which tells you there is something wrong with the “sounds” part of your pattern, and it requires tweaking

In-depth information

This section covers all the nitty-gritty details about noise recognition in general and about Parrot. It isn't required reading, but for those who are interested the information is compiled here in case they want to know more.

Power

The power value a measurement of loudness, a root-mean-square of the raw audio data, scaled up a bit to make it more readable in the debug logs.
This is really light and fast to calculate, which makes it perfect for something that runs at a steady heartbeat of 60 Hz, or every 16ish milliseconds.

This isn't an SI unit like dB, but its function is to be fast and readable, which it handles pretty well.

Format frequencies (F1 and F2) (https://en.wikipedia.org/wiki/Formant for in depth information)

As a measurement of pitch, Parrot calculates the first two formants of a sound as a guideline for how high or how low a sound is.
A naive estimation is made of these formants, to be given to the pattern matcher, which means they might not exactly match spectogram outputs you can view in programs like Audacity
F1 can be up to 1kHz, while F2 can fall between 800Hz up to 8kHz.

Sample rate

All the information Parrot needs of your voice can be captured at a sample rate of 16kHz, which is why the maximum we can determine is 8kHz
(https://en.wikipedia.org/wiki/Nyquist_frequency on why that is)
This does not mean however that higher frequencies aren't captured in the audio, but they do get distorted.

An example of this distortion is comparing a phone call with a CD recording. A hissing sound on a CD sounds really nice, and a lot less nice during a phone call, but you can still recognize whenever someone says the letter 'S'
The reason for choosing a lower sample rate, is that it requires a lot less processing than a higher sample rate.

The sample rate of a CD contains roughly 2.75 times more data to process than the 16kHz rate Parrot uses, while it doesn't really improve recognition of the model by all that much.

Latency and heart beat

Parrot optimizes for latency above all else, so we ideally use as little processing as possible to make the time from you uttering the sound, to Talon responding as fast as possible.

The last time I've checked the processing of single frame in Parrot.PY, it only took about a millisecond for it to reach the code that responds to the recognition.
Parrot also does not wait for the sound to be over before it starts recognizing it, it can start reacting to the onset of the sound right away if the patterns are tuned correctly.

With samples being taken every 15 milliseconds in the current model, you can get a response within 20 milliseconds of you starting the sound.