plot_viterbi

December 24, 2019

[ ]:
```
%matplotlib inline
```

1 Viterbi decoding

This notebook demonstrates how to use Viterbi decoding to impose temporal smoothing on frame-wise state predictions.

Our working example will be the problem of silence/non-silence detection.

[ ]:
```
# Code source: Brian McFee
# License: ISC

##################
# Standard imports
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
import librosa

import librosa.display
```

Load an example signal

[ ]:
```
y, sr = librosa.load('audio/sir_duke_slow.mp3')


# And compute the spectrogram magnitude and phase
S_full, phase = librosa.magphase(librosa.stft(y))


###################
# Plot the spectrum
plt.figure(figsize=(12, 4))
librosa.display.specshow(librosa.amplitude_to_db(S_full, ref=np.max),
                         y_axis='log', x_axis='time', sr=sr)
plt.colorbar()
plt.tight_layout()
```

As you can see, there are periods of silence and non-silence throughout this recording.

[ ]:
```python
# As a first step, we can plot the root-mean-square (RMS) curve
rms = librosa.feature.rms(y=y)[0]

times = librosa.frames_to_time(np.arange(len(rms)))

plt.figure(figsize=(12, 4))
plt.plot(times, rms)
plt.axhline(0.02, color='r', alpha=0.5)
plt.xlabel('Time')
plt.ylabel('RMS')
plt.axis('tight')
plt.tight_layout()

# The red line at 0.02 indicates a reasonable threshold for silence detection.
# However, the RMS curve occasionally dips below the threshold momentarily,
# and we would prefer the detector to not count these brief dips as silence.
# This is where the Viterbi algorithm comes in handy!
```

As a first step, we will convert the raw RMS score into a likelihood (probability) by logistic mapping

$P[V = 1|x] = \frac{\exp(x-\tau)}{1+\exp(x-\tau)}$

where x denotes the RMS value and $\tau = 0.02$ is our threshold. The variable V indicates whether the signal is non-silent (1) or silent (0).

We'll normalize the RMS by its standard deviation to expand the range of the probability vector

[ ]:
```python
r_normalized = (rms - 0.02) / np.std(rms)
p = np.exp(r_normalized) / (1 + np.exp(r_normalized))

# We can plot the probability curve over time:

plt.figure(figsize=(12, 4))
plt.plot(times, p, label='P[V=1|x]')
plt.axhline(0.5, color='r', alpha=0.5, label='Descision threshold')
plt.xlabel('Time')
plt.axis('tight')
plt.legend()
plt.tight_layout()
```

which looks much like the first plot, but with the decision threshold shifted to 0.5. A simple silence detector would classify each frame independently of its neighbors, which would result in the following plot:

[ ]:
```python
plt.figure(figsize=(12, 6))
ax = plt.subplot(2,1,1)
librosa.display.specshow(librosa.amplitude_to_db(S_full, ref=np.max),
                         y_axis='log', x_axis='time', sr=sr)
plt.subplot(2,1,2, sharex=ax)
plt.step(times, p>=0.5, label='Non-silent')
plt.xlabel('Time')
```

```
plt.axis('tight')
plt.ylim([0, 1.05])
plt.legend()
plt.tight_layout()
```

We can do better using the Viterbi algorithm. We'll use state 0 to indicate silent, and 1 to indicate non-silent. We'll assume that a silent frame is equally likely to be followed by silence or non-silence, but that non-silence is slightly more likely to be followed by non-silence. This is accomplished by building a self-loop transition matrix, where `transition[i, j]` is the probability of moving from state `i` to state `j` in the next frame.

[ ]:
```
transition = librosa.sequence.transition_loop(2, [0.5, 0.6])
print(transition)
```

Our `p` variable only indicates the probability of non-silence, so we need to also compute the probability of silence as its complement.

[ ]:
```
full_p = np.vstack([1 - p, p])
print(full_p)
```

Now, we're ready to decode! We'll use `viterbi_discriminative` here, since the inputs are state likelihoods conditional on data (in our case, data is rms).

[ ]:
```
states = librosa.sequence.viterbi_discriminative(full_p, transition)

# sphinx_gallery_thumbnail_number = 5
plt.figure(figsize=(12, 6))
ax = plt.subplot(2,1,1)
librosa.display.specshow(librosa.amplitude_to_db(S_full, ref=np.max),
                         y_axis='log', x_axis='time', sr=sr)
plt.xlabel('')
ax.tick_params(labelbottom=False)
plt.subplot(2, 1, 2, sharex=ax)
plt.step(times, p>=0.5, label='Frame-wise')
plt.step(times, states, linestyle='--', color='orange', label='Viterbi')
plt.xlabel('Time')
plt.axis('tight')
plt.ylim([0, 1.05])
plt.legend()
```

Note how the Viterbi output has fewer state changes than the frame-wise predictor, and it is less sensitive to momentary dips in energy. This is controlled directly by the transition matrix. A higher self-transition probability means that the decoder is less likely to change states.