

Speech Processing for Machine Learning: Filter banks, Mel-Frequency Cepstral Coefficients (MFCCs) and What's In-Between

Apr 21, 2016

Speech processing plays an important role in any speech system whether its Automatic Speech Recognition (ASR) or speaker recognition or something else. Mel-Frequency Cepstral Coefficients (MFCCs) were very popular features for a long time; but more recently, filter banks are becoming increasingly popular. In this post, I will discuss filter banks and MFCCs and why are filter banks becoming increasingly popular.

Computing filter banks and MFCCs involve somewhat the same procedure, where in both cases filter banks are computed and with a few more extra steps MFCCs can be obtained. In a nutshell, a signal goes through a pre-emphasis filter; then gets sliced into (overlapping) frames and a window function is applied to each frame; afterwards, we do a Fourier transform on each frame (or more specifically a Short-Time Fourier Transform) and calculate the power spectrum; and subsequently compute the filter banks. To obtain MFCCs, a Discrete Cosine Transform (DCT) is applied to the filter banks retaining a number of the resulting coefficients while the rest are discarded. A final step in both cases, is mean normalization.

Setup

For this post, I used a 16-bit PCM wav file from [here](#), called "OSR_us_000_0010_8k.wav", which has a sampling frequency of 8000 Hz. The wav file is a clean speech signal comprising a single voice uttering some sentences with some pauses in-between. For simplicity, I used the first 3.5 seconds of the signal which corresponds roughly to the first sentence in the wav file.

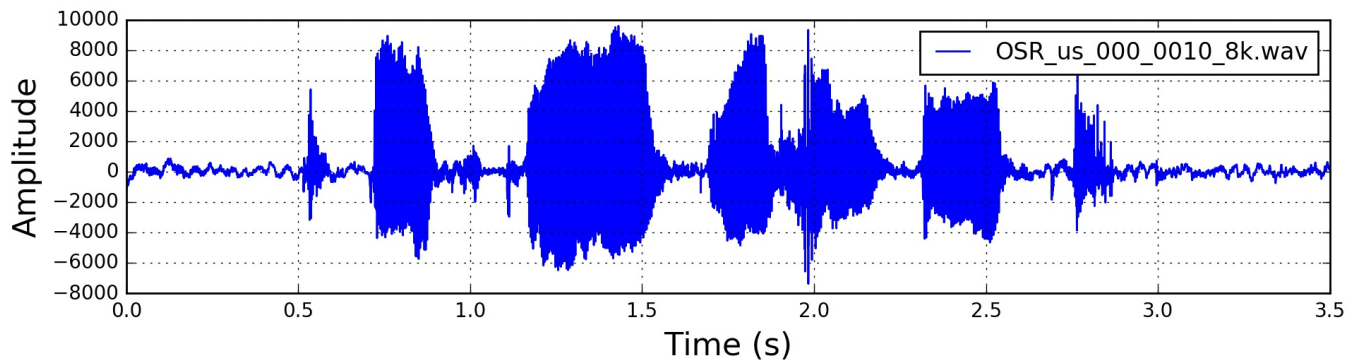
I'll be using Python 2.7.x, NumPy and SciPy. Some of the code used in this post is based on code available in this [repository](#).

```
import numpy
import scipy.io.wavfile
```

```
from scipy.fftpack import dct

sample_rate, signal = scipy.io.wavfile.read('OSR_us_000_0010_8k.wav') # File ass
signal = signal[0:int(3.5 * sample_rate)] # Keep the first 3.5 seconds
```

The raw signal has the following form in the time domain:



Signal in the Time Domain

Pre-Emphasis

The first step is to apply a pre-emphasis filter on the signal to amplify the high frequencies. A pre-emphasis filter is useful in several ways: (1) balance the frequency spectrum since high frequencies usually have smaller magnitudes compared to lower frequencies, (2) avoid numerical problems during the Fourier transform operation and (3) may also improve the Signal-to-Noise Ratio (SNR).

The pre-emphasis filter can be applied to a signal x using the first order filter in the following equation:

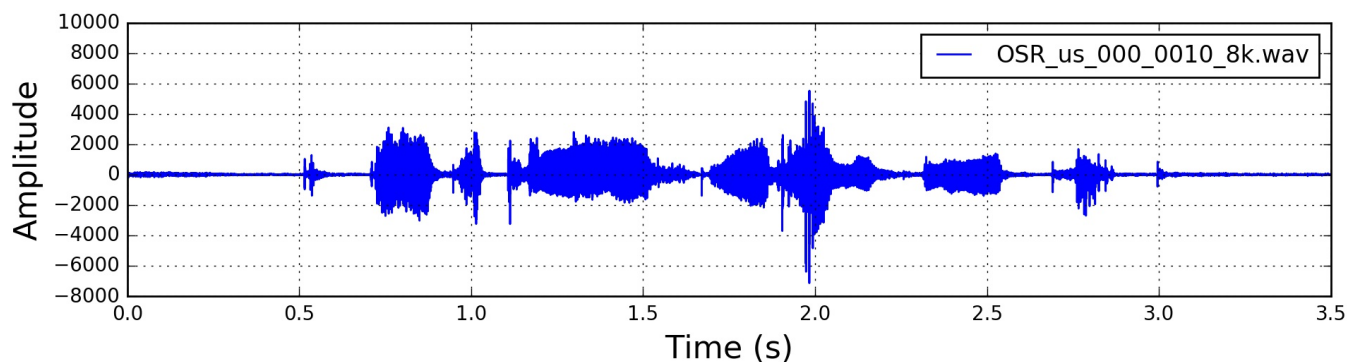
$$y(t) = x(t) - \alpha x(t - 1)$$

which can be easily implemented using the following line, where typical values for the filter coefficient (α) are 0.95 or 0.97, `pre_emphasis = 0.97`:

```
emphasized_signal = numpy.append(signal[0], signal[1:] - pre_emphasis * signal[:-1])
```

Pre-emphasis has a modest effect in modern systems, mainly because most of the motivations for the pre-emphasis filter can be achieved using mean normalization (discussed later in this post) except for avoiding the Fourier transform numerical issues which should not be a problem in modern FFT implementations.

The signal after pre-emphasis has the following form in the time domain:



Signal in the Time Domain after Pre-Emphasis

Framing

After pre-emphasis, we need to split the signal into short-time frames. The rationale behind this step is that frequencies in a signal change over time, so in most cases it doesn't make sense to do the Fourier transform across the entire signal in that we would lose the frequency contours of the signal over time. To avoid that, we can safely assume that frequencies in a signal are stationary over a very short period of time. Therefore, by doing a Fourier transform over this short-time frame, we can obtain a good approximation of the frequency contours of the signal by concatenating adjacent frames.

Typical frame sizes in speech processing range from 20 ms to 40 ms with 50% (+/-10%) overlap between consecutive frames. Popular settings are 25 ms for the frame size, `frame_size = 0.025` and a 10 ms stride (15 ms overlap), `frame_stride = 0.01`.

```
frame_length, frame_step = frame_size * sample_rate, frame_stride * sample_rate
signal_length = len(emphasized_signal)
frame_length = int(round(frame_length))
frame_step = int(round(frame_step))
num_frames = int(numpy.ceil(float(numpy.abs(signal_length - frame_length)) / fram

pad_signal_length = num_frames * frame_step + frame_length
z = numpy.zeros((pad_signal_length - signal_length))
pad_signal = numpy.append(emphasized_signal, z) # Pad Signal to make sure that al

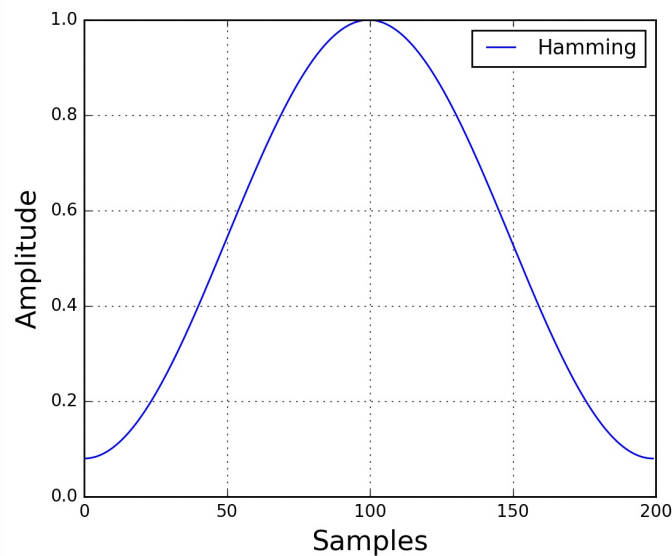
indices = numpy.tile(numpy.arange(0, frame_length), (num_frames, 1)) + numpy.tile
frames = pad_signal[indices.astype(numpy.int32, copy=False)]
```

Window

After slicing the signal into frames, we apply a window function such as the Hamming window to each frame. A Hamming window has the following form:

$$w[n] = 0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right)$$

where, $0 \leq n \leq N-1$, N is the window length. Plotting the previous equation yields the following plot:



Hamming Window

There are several reasons why we need to apply a window function to the frames, notably to counteract the assumption made by the FFT that the data is infinite and to reduce spectral leakage.

```
frames *= numpy.hamming(frame_length)
# frames *= 0.54 - 0.46 * numpy.cos((2 * numpy.pi * n) / (frame_length - 1)) # E
```

Fourier-Transform and Power Spectrum

We can now do an N -point FFT on each frame to calculate the frequency spectrum, which is also called Short-Time Fourier-Transform (STFT), where N is typically 256 or 512, **NFFT = 512**; and then compute the power spectrum (periodogram) using the following equation:

$$P = \frac{|FFT(x_i)|^2}{N}$$

where, x_i is the i^{th} frame of signal x . This could be implemented with the following lines:

```
mag_frames = numpy.absolute(numpy.fft.rfft(frames, NFFT)) # Magnitude of the FFT
pow_frames = ((1.0 / NFFT) * ((mag_frames) ** 2)) # Power Spectrum
```

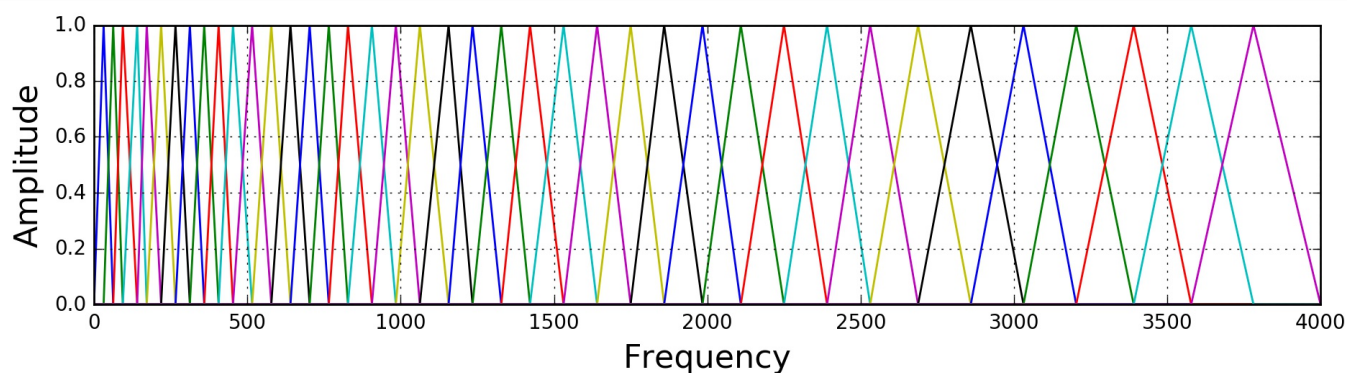
Filter Banks

The final step to computing filter banks is applying triangular filters, typically 40 filters, `nfilt = 40` on a Mel-scale to the power spectrum to extract frequency bands. The Mel-scale aims to mimic the non-linear human ear perception of sound, by being more discriminative at lower frequencies and less discriminative at higher frequencies. We can convert between Hertz (f) and Mel (m) using the following equations:

$$m = 2595 \log_{10}\left(1 + \frac{f}{700}\right)$$

$$f = 700(10^{m/2595} - 1)$$

Each filter in the filter bank is triangular having a response of 1 at the center frequency and decrease linearly towards 0 till it reaches the center frequencies of the two adjacent filters where the response is 0, as shown in this figure:



Filter bank on a Mel-Scale

This can be modeled by the following equation (taken from here):

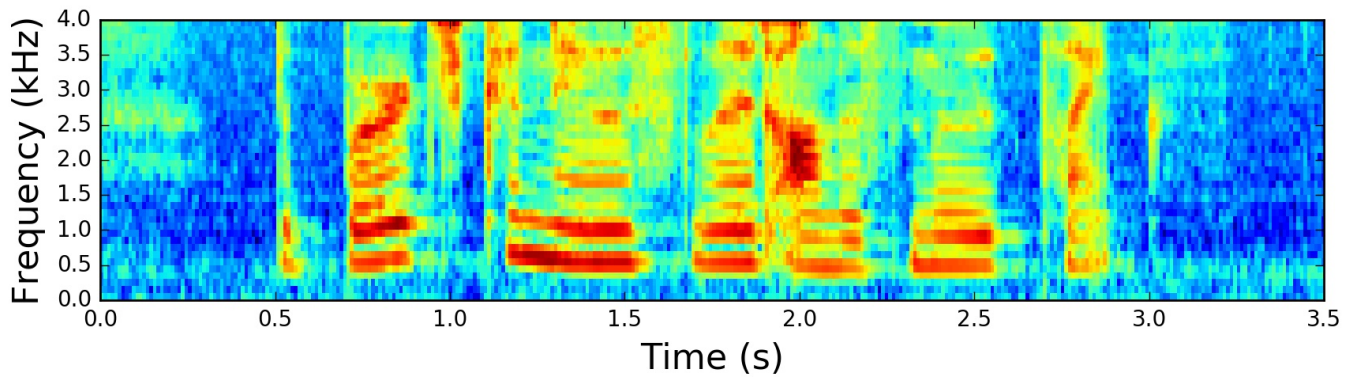
$$H_m(k) = \begin{cases} 0 & k < f(m-1) \\ \frac{k - f(m-1)}{f(m) - f(m-1)} & f(m-1) \leq k < f(m) \\ 1 & k = f(m) \\ \frac{f(m+1) - k}{f(m+1) - f(m)} & f(m) < k \leq f(m+1) \\ 0 & k > f(m+1) \end{cases}$$

```
low_freq_mel = 0
high_freq_mel = (2595 * numpy.log10(1 + (sample_rate / 2) / 700)) # Convert Hz to Mel
mel_points = numpy.linspace(low_freq_mel, high_freq_mel, nfilt + 2) # Equally spaced Mel points
hz_points = (700 * (10**(mel_points / 2595) - 1)) # Convert Mel to Hz
bin = numpy.floor((NFFT + 1) * hz_points / sample_rate)

fbank = numpy.zeros((nfilt, int(numpy.floor(NFFT / 2 + 1))))
for m in range(1, nfilt + 1):
    f_m_minus = int(bin[m - 1]) # left
    f_m = int(bin[m]) # center
    f_m_plus = int(bin[m + 1]) # right

    for k in range(f_m_minus, f_m):
        fbank[m - 1, k] = (k - bin[m - 1]) / (bin[m] - bin[m - 1])
    for k in range(f_m, f_m_plus):
        fbank[m, k] = (bin[m + 1] - k) / (bin[m + 1] - bin[m])
filter_banks = numpy.dot(pow_frames, fbank.T)
filter_banks = numpy.where(filter_banks == 0, numpy.finfo(float).eps, filter_banks)
filter_banks = 20 * numpy.log10(filter_banks) # dB
```

After applying the filter bank to the power spectrum (periodogram) of the signal, we obtain the following spectrogram:



Spectrogram of the Signal

If the Mel-scaled filter banks were the desired features then we can skip to mean normalization.

Mel-frequency Cepstral Coefficients (MFCCs)

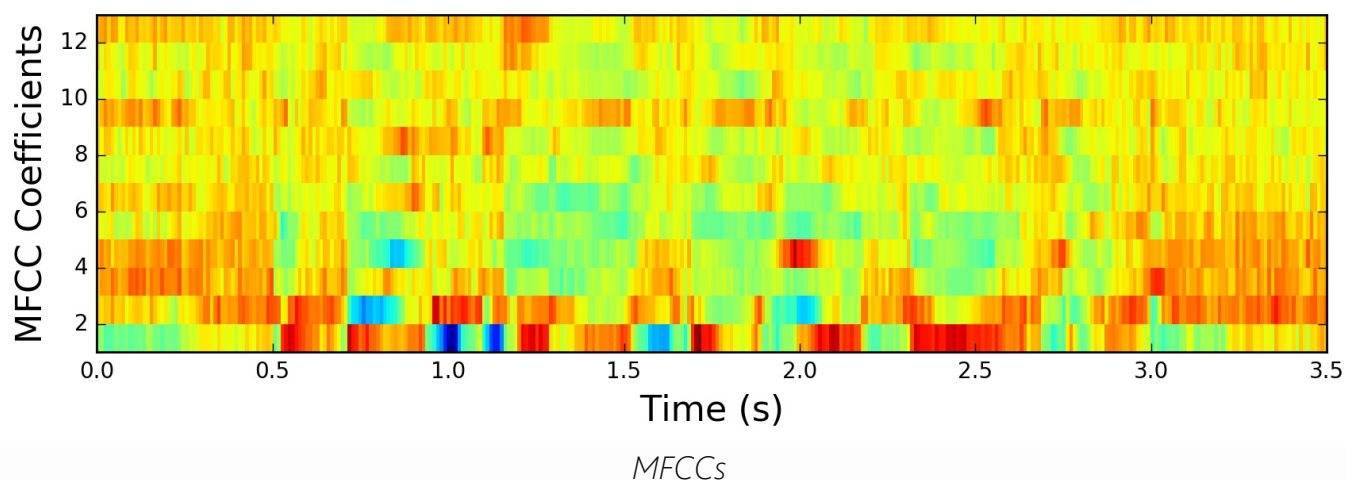
It turns out that filter bank coefficients computed in the previous step are highly correlated, which could be problematic in some machine learning algorithms. Therefore, we can apply Discrete Cosine Transform (DCT) to decorrelate the filter bank coefficients and yield a compressed representation of the filter banks. Typically, for Automatic Speech Recognition (ASR), the resulting cepstral coefficients 2-13 are retained and the rest are discarded; `num_ceps = 12`. The reasons for discarding the other coefficients is that they represent fast changes in the filter bank coefficients and these fine details don't contribute to Automatic Speech Recognition (ASR).

```
mfcc = dct(filter_banks, type=2, axis=1, norm='ortho')[:, 1 : (num_ceps + 1)] # K
```

One may apply sinusoidal liftering¹ to the MFCCs to de-emphasize higher MFCCs which has been claimed to improve speech recognition in noisy signals.

```
(nframes, ncoeff) = mfcc.shape
n = numpy.arange(ncoeff)
lift = 1 + (cep_lifter / 2) * numpy.sin(numpy.pi * n / cep_lifter)
mfcc *= lift #*
```

The resulting MFCCs:

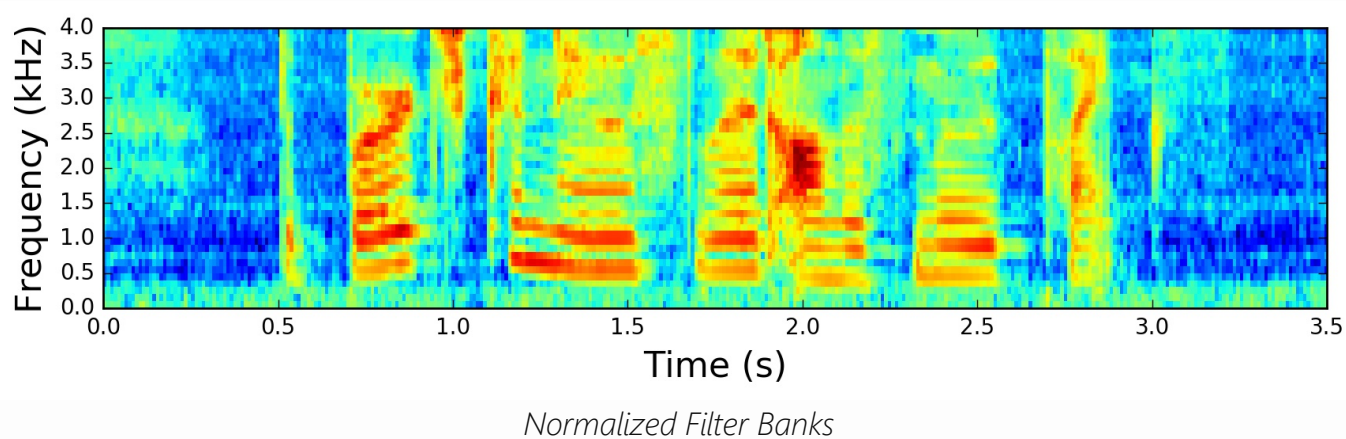


Mean Normalization

As previously mentioned, to balance the spectrum and improve the Signal-to-Noise (SNR), we can simply subtract the mean of each coefficient from all frames.

```
filter_banks -= (numpy.mean(filter_banks, axis=0) + 1e-8)
```

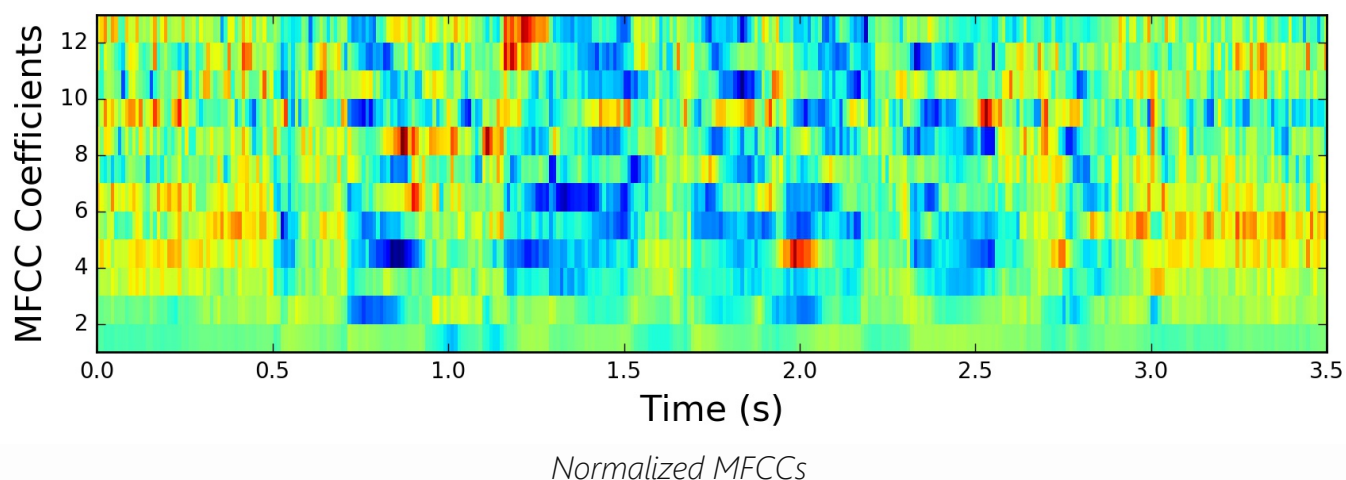
The mean-normalized filter banks:



and similarly for MFCCs:

```
mfcc -= (numpy.mean(mfcc, axis=0) + 1e-8)
```

The mean-normalized MFCCs:



Filter Banks vs MFCCs

To this point, the steps to compute filter banks and MFCCs were discussed in terms of their motivations and implementations. It is interesting to note that all steps needed to compute filter banks were motivated by the nature of the speech signal and the human perception of such signals. On the contrary, the extra steps needed to compute MFCCs were motivated by the limitation of some machine learning algorithms. The Discrete Cosine Transform (DCT) was needed to decorrelate filter bank coefficients, a process also referred to as whitening. In particular, MFCCs were very popular when Gaussian Mixture Models - Hidden Markov Models (GMMs-HMMs) were very popular and together, MFCCs and GMMs-HMMs co-evolved to be the standard way of doing Automatic Speech Recognition (ASR)². With the advent of Deep Learning in speech systems, one might question if MFCCs are still the right choice given that deep neural networks are less susceptible to highly correlated input and therefore the Discrete Cosine Transform (DCT) is no longer a necessary step. It is beneficial to note that Discrete Cosine Transform (DCT) is a linear transformation, and therefore undesirable as it discards some information in speech signals which are highly non-linear.

It is sensible to question if the Fourier Transform is a necessary operation. Given that the Fourier Transform itself is also a linear operation, it might be beneficial to ignore it and attempt to learn directly from the signal in the time domain. Indeed, some recent work has already attempted this and positive results were reported. However, the Fourier transform operation is a difficult operation to learn and may arguably increase the amount of data and model complexity needed to achieve the same performance. Moreover, in doing Short-Time Fourier Transform (STFT), we've assumed the signal to be stationary within this short time and therefore the linearity of the Fourier transform would not pose a critical problem.

Conclusion

In this post, we've explored the procedure to compute Mel-scaled filter banks and Mel-Frequency Cepstrum Coefficients (MFCCs). The motivations and implementation of each step in the procedure were discussed. We've also argued the reasons behind the increasing popularity of filter banks compared to MFCCs.

tl;dr: Use Mel-scaled filter banks if the machine learning algorithm is not susceptible to highly correlated input. Use MFCCs if the machine learning algorithm is susceptible to correlated input.

Citation:

```
@misc{fayek2016,  
  title   = "Speech Processing for Machine Learning: Filter banks, Mel-Frequency  
  author  = "Haytham M. Fayek",  
  year    = "2016",  
  url     = "https://haythamfayek.com/2016/04/21/speech-processing-for-machine-le  
}
```

1. Liftering is filtering in the cepstral domain. Note the abuse of notation in *spectral* and *cepstral* with *filtering* and *liftering* respectively. [↗](#)
2. An excellent discussion on this topic is in [this thesis](#). [↗](#)

Share: [Facebook](#) | [LinkedIn](#) | [Pinterest](#) | [Reddit](#) | [Twitter](#)

102 Comments HaythamFayek  Disqus' Privacy Policy

 1 Login ▾

 Recommend 32

 Tweet

 Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name



Andrea Argudo • 3 years ago • edited

Hi, Haytham Fayek, amazing article!. I have a question, why do you only use the hamming windows instead of others? I mean, there is a big reason?. Thanks in advance.

4 ^ | v 1 • Reply • Share ›



Tomas Garcia ➔ Andrea Argudo • 2 years ago • edited

hamming window has been historically the standard for speech processing. It provides a good balance between frequency resolution (i.e. the separability of closely grouped peaks in the frequency domain) and dynamic range (i.e. the size of the main lobe compared to side lobes) that works well for speech processing applications. square-root hann is also common for applications where the signal is being reconstructed as it is more forgiving when applying aggressive frequency domain processing.

2 ^ | v • Reply • Share ›



Haytham Fayek Mod ➔ Andrea Argudo • 2 years ago

Hi Andrea, Thanks! I don't have a solid answer to this question. I don't think there is a big reason to use the hamming window particularly, but for example, as opposed to the Hann window, the Hamming window's settings cancel the largest side lobe which results in lower error. Whether this has an effect on the end result is unclear to me.

^ | v • Reply • Share ›



chandra sutrisno • 3 years ago

Hi,

Thank you for this useful tutorial. Currently, I am trying to build voice/speaker recognition so this is really helpful.

I use your method to extract the features from my data set. I have 10 sample voice and each sample yields different shape of array? Is this real? How come this can happen? And then how to feed it into machine learning model since it's a 2D array? Should I use any NN model?

Please advise

2 ^ | v • Reply • Share ›



Haytham Fayek Mod ➔ chandra sutrisno • 2 years ago

Yes, since your voice samples are of different lengths, the number of features would vary according to the lengths. You'll want to divide those into equal number of frames and then align the transcription to the frames.

^ | v • Reply • Share ›



שחר נחמיאס ➔ chandra sutrisno • 3 years ago

Would like to hear the answer to that questions as well.
Great tutorial!

^ | v • Reply • Share ›



Asheesh Sharma • 2 years ago

Hi, Haytham,

Thank you for sharing your knowledge.

Given that my interest in ASR is relatively new, MFCCs had me confused, which often raised the doubts you mentioned in the Filter Banks Vs MFCCs discussion.

As you have mentioned, Fast Fourier transforms are linear operations. Other than that, one should also consider the obvious computational overhead it brings in with itself (for a serious application).

Unfortunately, I couldn't find any research papers which show the positive results of "directly learning from the input signal". I might also have misunderstood what you meant there (i.e signal in time -> MFCCs). Can you point me in the right direction?

Currently, I am using filter-banks as a result of the dot product b/w the signal frames and filter banks directly, followed by DCT. I have yet to compare the results, but do you think that the approach is flawed?

with regards,
Asheesh

2 ^ | v 1 • Reply • Share ›



Haytham Fayek Mod ➔ Asheesh Sharma • 9 months ago • edited

Thanks! I am not sure why would it be flawed unless if I've misunderstood your comment.

PS. FFT is a very efficient and fast operation.

1 ^ | v • Reply • Share ›



Pratyush Rath • 10 months ago

Hi , Haytham Fayek , great article . I am using this article for making a project . The project is to create Baby Cry detection algorithm . I am no able to create the image of the final MFCC array which is (samples_per_frame * MFCC_features)

Please help . I am stuck at this step for quite long .

1 ^ | v • Reply • Share ›



Haytham Fayek Mod ➔ Pratyush Rath • 9 months ago • edited

Thanks! I think it should be number_of_frames * MFCC_features.

^ | v • Reply • Share ›



Vaishali Rajput • 21 days ago

what is cep_lifter? in one of the steps it is mentioned

^ | v • Reply • Share ›



sadegh soleimani • 2 months ago

Hi, thanks for the article , this semester I have sound processing in medical prognosis and it helped me a lot I would be appreciative to have addresses of other articles in this area that you may know or write about like pitch detection, pitch tracking, lpc and so on
thank you in advance and thank you for precise information and explanation

^ | v • Reply • Share ›



sadegh soleimani → sadegh soleimani • 2 months ago • edited

and one Question

Do you use this algorithm cause i didn't see the $\log(\text{abs()}^{**2})$ part of it



^ | v • Reply • Share ›



Hind Ait Mait • 3 months ago

thank you soo much sir. I have a question, if we applied the delta mfccs, and we detected their maximum local how we can use it for determining the boundaries of signal segments ?

^ | v • Reply • Share ›



Fanzu_U • 4 months ago

Hi, **@Haytham Fayek**

Thank you for this useful tutorial. I tried your step by step and then, "name 'floor' is not defined". i didn't see floor declaration. what it is 'floor' ?

^ | v • Reply • Share ›



Madeline Foltynova • 4 months ago

Hi, how would you implement this if I have a whole folder full of .wav files and I need to make this spectrogram to all of them, so that I have one for each .wav sound.

^ | v • Reply • Share ›



Wouter Vandenputte • 4 months ago

Hey, I followed everything step by step but I had a question with all the units. If for example this would be a random windowed frame





[see more](#)

^ | ▾ • Reply • Share ›



Jim_Lewis • 6 months ago • edited

Hi, Haytham Fayek. Was wondering why filter_banks are calculated as:
`filter_banks = numpy.dot(pow_frames, fbank.T)?`

In edX.org DEV287, speech signal processing module, we worked with the magnitude spectrum instead of the power spectrum but our equivalent of filter_banks, which we called fbank, was calculated roughly as:

```
fbank = numpy.dot(mel_filterbank, magspec) # am leaving out object references
```

Starting with a mel filterbank that was 40 x 257 and a speech signal that after processing and rfft transformation gave a 257 x 584 array, the dot product produced a 40 x 584 log mel frequency spectrogram ndarray.

after padding against zero values and taking the natural log of the ndarray values, we were instructed to perform feature normalization by calculating the mean frame vector and subtracting it from all frames.

Given the dimensions of our log mel spectrogram of 40 x 584 (we wanted the mean of the column vectors to subtract), I found the following worked best (a takeoff on your equation but using keepdims for broadcasting protection on axis 1, given the shape of the array I was

[see more](#)

^ | ▾ • Reply • Share ›



Luke Y Choi → Jim_Lewis • 4 months ago

hi Jim,
what did you mean by "padding against zero value..."? does it means non-zero padding (i.e., padding non-zero value?)

^ | ▾ • Reply • Share ›



Haytham Fayek Mod → Jim_Lewis • 5 months ago

Hi Jim, if I've followed your steps correctly, I think both ways will lead to the same results except for the difference between magnitude and power.

^ | ▾ • Reply • Share ›

^ | v • Reply • Share ›



Luke Y Choi → Haytham Fayek • 4 months ago

hi Haytham Fayek,
have you ever seen any paper or speech recog engine, available, using the non-zero padding instead of zero-padding?

^ | v 1 • Reply • Share ›



Jim_Lewis • 6 months ago

Great article! Fills in the blanks on the details of Speech Signal Processing that Microsoft left out in Module 02 of its DEV287x course on edX. Although the edX.org course is good, I understand the whole process so much better after reading your article.

^ | v • Reply • Share ›



Haytham Fayek Mod → Jim_Lewis • 5 months ago

Thanks!

^ | v • Reply • Share ›



Jacek • 7 months ago

Hi, you missed `cep_lifter` declaration. What it is?
Great article!

^ | v • Reply • Share ›



George → Jacek • 6 months ago

default value is 22

<https://github.com/jameslyo...>

1 ^ | v • Reply • Share ›



Haytham Fayek Mod → Jacek • 5 months ago • edited

Thanks. `cep_lifter` is 22.

^ | v • Reply • Share ›



George Marios Papasotiriou • 8 months ago

Hi Haytham,

This is an amazing and detailed explanation - thanks for sharing!

I am a little confused with the step `'filter_banks = numpy.dot(pow_frames, fbank.T)'`

My `pow_frames` have a different dimension to my `fbank` vector. My `fbank` has 40x257 due to the 40 filters and NFFT being 512.

However, my `pow_frames` 200x2000 as the signal is split into 2000 frames each of 200 data points.

How would you recommend I approach this issue?

^ | v • Reply • Share ›



Haytham Fayek Mod → George Marios Papasotiriou • 5 months ago

Thanks! The dimension of `pow_frames` should also be 257 after taking a 512-point fft.

^ | v • Reply • Share ›



George Marios Papasotiriou → George Marios Papasotiriou • 8 months ago • edited

The frames have the aforementioned dimensions because I chose for frames of 25ms with 10ms overlap, whilst the signal is a 30second fragement of the same signal you used as your input.

^ | v • Reply • Share ›



John Black • 9 months ago

Hi Haytham. I really enjoyed reading the article. I'm curious about one thing. You've mentioned about the problem that some machine learning algorithms are more susceptible to highly correlated input than other. Can you elaborate on that? What ML algorithms did you mean?

^ | v • Reply • Share ›



Kamil Dabek • a year ago

Hi,

Great article. Could you explain what you mean by highly correlated input. Input of what kind and how correlated?

Best Regards,
KD

^ | v • Reply • Share ›



Haytham Fayek Mod → Kamil Dabek • 9 months ago

Thanks! The Mel frequency spectral coefficients are correlated with each other as the filter banks producing these coefficients are overlapping.

1 ^ | v • Reply • Share ›



jpf27 • a year ago

Hi Haytham, thank you for this thorough and helpful article. If you have a reference for the claim that sinusoidal liftering improves speech recognition, I would love to read it.

^ | v • Reply • Share ›



Haytham Fayek Mod → jpf27 • 9 months ago

Thanks! I'd be surprised if sinusoidal liftering significantly improves neural network based speech recognition.

^ | v • Reply • Share ›



Eduardo G.R. • a year ago

Very interesting article! Can you please explain why you didn't divide by the standard

deviation after mean subtraction in your normalization method? It's confusing to me.

^ | v • Reply • Share ›



Haytham Fayek Mod ➔ Eduardo G.R. • 9 months ago

Thanks! You can divide by the standard deviation.

1 ^ | v • Reply • Share ›



Eslam Ahmed • a year ago

Hello, I am working on a speaker recognition project and I have a problem, if any one can help, please do.

I have already implemented all the stages for this project as following (capture audio from mic - apply emphasize filter - framing with overlapping - silence removal - apply hamming window - apply reordering for FFT using Bit reversal - Apply FFT - get magnitude - Get power spectral - MFCC this include creation of the filterbank and multiply it by the signal then sum each filter bank then get the log of the sum for base 10 and finally DCT). the final DCT output is a 2D array of [NumberOfFrames,NumberOfCoefficients] for each audio clip of 3 seconds, this 2D array is saved as it carries the unique features of the speaker, during the training mode a three of 2D arrays are saved as training templates for each person as each person enters a 3 records, one 2D array for each record . during the test mode the same sequence is done and it also produced a one 2D array as the output of the final DCT, the euclidean distance is measured between this one which created at test mode and each one of the saved arrays during the training mode then a voting is done for the nearest 3 templates to determine the dominant person across the 3 nearest. the problem is the bad accuracy even increasing the number of training data for each person , I have searched the internet for what is exactly should be done after the mfcc stage but no obvious or full tutorial. I wrote the code from scratch in c# for desktop application and in c language for further use in embedded application this is the my main target. there are not any ready made software libraries. all above stages have been passed through a unit testing , every line and variable of code has been traced carefully and finally all passed through integration test until producing the MFCC coefficients and every thing seems to be okay. I do not want a ready code, but I want a clean tutorial to understand what is exactly the optimal step should be done after MFCC to finalize the project with good accuracy of identification.

^ | v • Reply • Share ›



PMacedoFilho Filho • a year ago

Many thanks Haytham. Excellent explanation on a theme not so easy. I made a comparison with the results with the results obtained using the python_speech_features module and the MFCC coefficients are very different. You can tell the reason. thank you !!!

^ | v • Reply • Share ›




Rob Webster • a year ago

Hi Haytham Fayek, fantastic article. This helped me out so much with my dissertation, I was wondering if it would be okay to have you in my "acknowledgements" section?

^ | v • Reply • Share ›



Haytham Fayek Mod ➔ Rob Webster • a year ago

 Of course, Thank you! I am glad it helped.

^ | v • Reply • Share ›



Robin Algayres • 2 years ago

Hi Fayek, great article !

I don't understand why DCT would discard information. DCT is invertible so it should not lose anything from the signal by applying it to the melfilters. Same question for the FFT.

Thank you!

^ | v • Reply • Share ›



Haytham Fayek Mod ➔ Robin Algayres • a year ago

Thanks, Robin! In the FFT, we discard the phase information, while in the DCT, we only keep the first n coefficients and discard the remaining ones; so in both cases, we're throwing away information.

1 ^ | v • Reply • Share ›



Julio • 2 years ago

how do you plot mfcc and pow_frame?

^ | v • Reply • Share ›



Ego ren ➔ Julio • 2 years ago

Hello, I use these following code to get plot pic.

```
plt.subplot(312)
filter_banks -= (numpy.mean(filter_banks,axis=0) + 1e-8)
plt.imshow(filter_banks.T, cmap=plt.cm.jet, aspect='auto')
plt.xticks(np.arange(0, (filter_banks.T).shape[1],
int((filter_banks.T).shape[1] / 4)),
['0s', '0.5s', '1s', '1.5s', '2.5s', '3s', '3.5'])
ax = plt.gca()
ax.invert_yaxis()
plt.title('the spectrum image')
```

7 ^ | v • Reply • Share ›



Y V S Harish ➔ Ego ren • a month ago • edited

it should be "/" 7" right ?

^ | v • Reply • Share ›



Alex Beloi • 2 years ago

Nice write-up!

I found a small typo: "Fourier transform across the entire signal in that we would loose the frequency contours", I believe you mean "lose" instead of "loose".

^ | v • Reply • Share ›



Haytham Fayek Mod ➔ Alex Beloi • a year ago

Thanks Alex! Fixed.

^ | v • Reply • Share ›



Shreyas Gupta • 2 years ago • edited

There is a small mistake in the frames calculation. with `np.ceil`, we will loose one frame in some cases. This is loss of some signal data. I don't know whether it matters but we can overcome that with We can overcome this with padding the audio signal with zeroes and then calculating number of frames by

```
int(np.floor((signal_length - frame_length) / frame_step) + 1)
```