Working with ECG — Heart Rate data, on Python

Bartek Kulas · Follow
9 min read · Feb 8, 2023

An introduction to working with heart rate data (ECG) on Python

Over the past 5–10 years, we have seen a massive boom in the prominence of popular bio-wearables in the consumer space such as the Oura ring, Apple watch, Garmin, and Fitbit. Continuous bio-sensor data, such as electrocardiogram (ECG), are now more accessible than ever before.

We will cover the following:

- What is Heart Rate Variability?
- Template matching and peak detection
- Signal cleaning
- Time-domain features
- Frequency-based features

Background

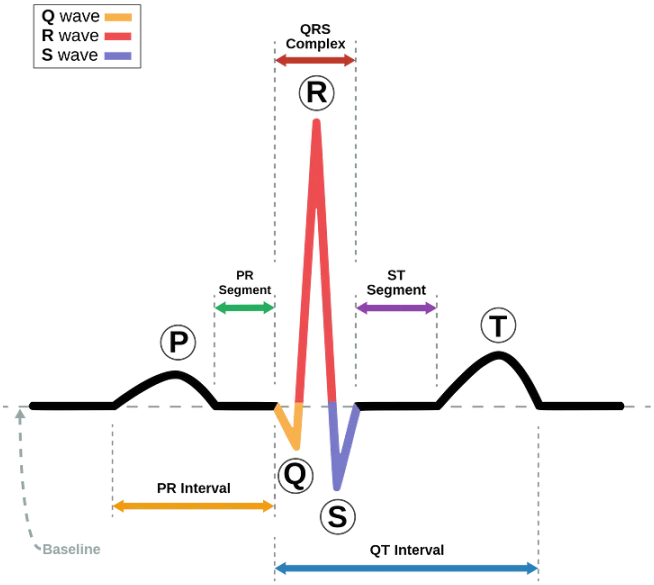
Before any code or processing steps, lets touch on some background

The cardiac cycle

ECG (electrocardiogram) records the electrical signals in the heart.

A typical ECG recording has a cyclical rhythm pattern consisting of: **P, Q, R, S**

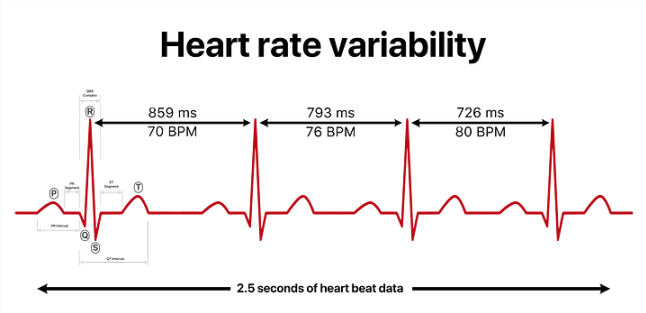
and **T** phases. The most striking part of the cycle is the **QRS complex** which is associated with depolarization of the hearts ventricles (and the ejection of blood from the heart). For todays purposes, we are particularly interested in the R-peak.



https://en.wikipedia.org/wiki/Electrocardiography

R-R Intervals

Finding the exact R-peak locations allows you to calculate intervals between beats. These are known as **R-R intervals**, which are then used to calculate indices of **Heart Rate Variability (HRV).**



https://en.wikipedia.org/wiki/Heart_rate_variability

> *Each R-R interval can be converted into beats per minute (BPM) by a simple calculation, i.e: 60 / .855 = 70 BPM*

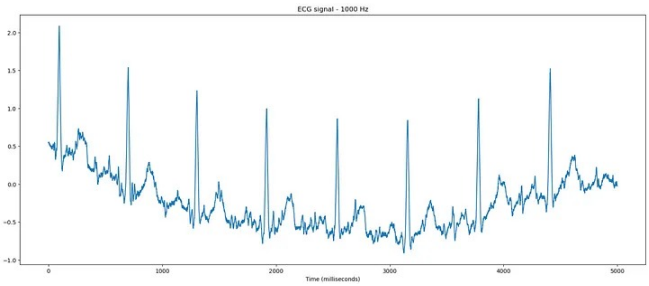
Working with our ECG data

And that, is what a textbook ECG waveforms looks like. Now, let's import some (noisy) data from the real world, that I recorded using a lead-2 ECG device.

Start by importing the raw data using pandas, and plotting a quick data examination:

```
# Import data
path = r'./data/sample_file_ECG.csv'
df = pd.read_csv(path)
print(df)

# plot
plt.title("ECG signal - 1000 Hz")
plt.plot(df.ecg[0:5000])
plt.xlabel('Time (milliseconds)')
plt.show()
```



This sample has a relatively clean 5-second period of ECG signal. Some signal drift (U-shape) can be seen, perhaps due to movement or respiration.

Note: in this case, the lead-2 ECG sensor records in voltage units (mV)

Template Matching

Before proceeding with R-wave peak detection, lets amplify the peaks of our QRS complex, to separate this feature from the rest of the signal, and improve detection! A simple method is to apply **filtering** using **template matching.**

Ideally, we should select a filter that generalizes well across all sorts of variants found in real world data. One option is to develop a complex model for matching QRS segments, think **CNN** — where the neural network can learn effective filters for certain features automatically using back propagation. However, lets not get too fancy here and save that for another time… for simplicity's sake, we can implement a **sinewave filter**:

```
# linear spaced vector between 0.5 pi and 1.5 pi
v = np.linspace(0.5 * np.pi, 1.5 * np.pi, 15)

# create sine filter for approximating QRS feature
peak_filter = np.sin(v)

# compute cross correlation between ecg and the sine filter
ecg_transformed = np.correlate(df.ecg, peak_filter, mode="same")

# and plot the raw + filtered signal
plt.figure(figsize=(20,8))
plt.title('ECG signal - 1000 Hz')
plt.plot(ecg_transformed[0:5000], alpha = 0.8, c='orange')
plt.plot(df.ecg[0:5000], alpha = 1)
plt.gca().legend(('filtered','raw signal'))
plt.xlabel('Time (milliseconds)')
plt.show()
```

> *The filtered signal (blue) is the computed **cross-correlation** between the raw signal (orange) and the sinewave filter. This is often referred to as a **convolution**. You can think of this as a measure of similarity between two signals. In our plot, we easily observe distinct peaks since the R-wave feature is prominent and easily separable—for robust detection of more intricate features, especially those buried under background noise, we would likely require a more complex and specific transform algorithm*

Peak Detection

Now that we have successfully amplified the QRS feature and created some

separation from the rest of the signal, we can move on to detecting and labeling the R-R peaks in our data!

```
# calculates the differences between consecutive samples
diff_sig_ecg = np.diff(df['ECG'])

rr_peaks, _ = find_peaks(ecg_transformed, distance=1000*(30/60))
plt.plot(ecg_transformed, alpha = 0.8)
plt.scatter(rr_peaks, ecg_transformed[rr_peaks], color='red')
plt.xlim(0,10000)
plt.title("ECG signal - 1000 Hz")
plt.show()
```

Looking good. The red-dot labels clearly show the peak tracking is precise. For a quick glimpse of the full-length data, lets create two more plots:

1. R-R intervals over time (seconds), and
2. An interpolated function for smoothing:

```
# compute the diff along the time axis to end up with the R-R intervals
rr_ecg = np.diff(rr_peaks)

# fit function to the dataset
x_ecg = np.cumsum(rr_ecg)/1000
f_ecg = interp1d(x_ecg, rr_ecg, kind='cubic', fill_value= 'extrapolate')

# sample rate for interpolation
fs = 4
steps = 1 / fs

# sample using the interpolation function
xx_ecg = np.arange(0, np.max(x_ecg), steps)
rr_interpolated_ecg = f_ecg(xx_ecg)

plt.subplot(211)
plt.title('rr-intervals')
plt.plot(x_ecg, rr_ecg, color='k', markerfacecolor='#A999D1',marker='o', markers
plt.ylabel('rr-interval (ms)')

plt.subplot(212)
plt.title('rr-intervals (cubic interpolation)')
plt.plot(xx_ecg, rr_interpolated_ecg, color='r')
plt.xlabel('Time (s)')
plt.ylabel('RR-interval (ms)')

plt.show()
```

That doesn't look too bad. A quick look tells us the mean R-R intervals is around 650 ms, or, about 92 bpm. Perhaps, it appears we have a few potential outliers in our data, or inaccurate peak detections (the intervals around ~750ms and the final peak)

Cleaning & dealing with outliers

This step requires careful evaluation, and depends on the precision and

reliability needs of your signal detection.

For demonstration, pretend we are confident that there were a few false peak detections. We select these 'suspicious' points using z-score thresholding, followed by replacement with the data median:

```
rr_ecg[np.abs(zscore(rr_ecg)) > 2] = np.median(rr_ecg)

x_ecg = np.cumsum(rr_ecg)/1000
f_ecg = interp1d(x_ecg, rr_ecg, kind='cubic', fill_value= 'extrapolate')

xx_ecg = np.arange(0, np.max(x_ecg), steps)
clean_rr_interpolated_ecg = f_ecg(xx_ecg)

plt.figure(figsize=(25,5))
plt.title('Error using z-score')
plt.plot(rr_interpolated_ecg)
plt.plot(clean_rr_interpolated_ecg)
plt.xlabel('Time (s)')
plt.ylabel('RR-interval (ms)')
plt.show()
```

> ***Remember**: the more preprocessing steps applied to the raw data, the more artificial the signal becomes, and you begin running an increased risk of removing 'real' features from the data that may distort your findings. Sometimes less is more*

Heart Rate Variability

You may have heard of **Heart Rate Variability** (**HRV**) before, which is the continuous up-and-down variability of the R-R intervals over time. HRV is a reflection of the adaptability, and the ever-changing oscillations of your heart

There are many commonly used methods of quantifying HRV, but in general, a higher index of HRV is a sign of a healthy functioning heart. For example, consumer health-apps and products may indicate that you are 'less stressed' and 'happier', when detecting a high HRV score.

Now, do not stress, because these HRV metrics are straight forward to calculate!

Time-domain analysis

Using only *rr_ecg* we can explore some of the commonly used time-domain metrics to quantify HRV:

```
def timedomain(rr):
    results = {}

    hr = 60000/rr

    # HRV metrics
    results['Mean RR (ms)'] = np.mean(rr)
    results['STD RR/SDNN (ms)'] = np.std(rr)
    results['Mean HR (Kubios\' style) (beats/min)'] = 60000/np.mean(rr)
    results['Mean HR (beats/min)'] = np.mean(hr)
    results['STD HR (beats/min)'] = np.std(hr)
    results['Min HR (beats/min)'] = np.min(hr)
    results['Max HR (beats/min)'] = np.max(hr)
    results['RMSSD (ms)'] = np.sqrt(np.mean(np.square(np.diff(rr))))
    results['NN50'] = np.sum(np.abs(np.diff(rr)) > 50)*1
    results['pNN50 (%)'] = 100 * np.sum((np.abs(np.diff(rr)) > 50)*1) / len(rr)
    return results

timedomain(rr_ecg)
```

The calculation methods and applications of these metrics vary. For example, the **RMSSD** (*root mean square of successive differences*) is computed from the differences between successive heartbeats, and the metric has a strong backbone of research in the Autonomic Nervous System, as well clinical associations with diseases such as cardiac health and epilepsy.

For a deeper dive, this review describes the validity and assessment strategies for heart rate and HRV metrics in great detail. As a summary:

- **STD RR/SDNN**: Often calculated over a 24-hour period. SDNN reflects all the cyclic components responsible for variability, therefore it represents total variability
- **RMSSD:** square root of the mean of the squares of the successive differences between adjacent NNs
- **STD**: The standard deviation of the successive differences between adjacent NNs
- **NNxx**: the number of pairs of successive NNs that differ by more than xx ms (we used 50 ms in our example)
- **pNNxx**: the proportion of NNxx divided by total number of NNs

Frequency-domain features

Additionally, the signal can be decomposed into its frequency components. Using the interpolated ECG data and Welch's method from the scipy.signal module, we can estimate power spectral density:

```
fxx, pxx = signal.welch(x=clean_rr_interpolated_ecg, fs=fs, nperseg=256)

# fit a function for plotting bands
powerspectrum_f = interp1d(fxx, pxx, kind='cubic', fill_value= 'extrapolate')

plt.figure(figsize=(15,6))
plt.title("FFT Spectrum (Welch's periodogram)")

# setup frequency bands for plotting
x_VLF = np.linspace(0, 0.04, 100)
x_LF = np.linspace(0.04, 0.15, 100)
x_HF = np.linspace(0.15, 0.4, 100)

plt.gca().fill_between(x_VLF, powerspectrum_f(x_VLF), alpha=0.5, color="#F5866F"
plt.gca().fill_between(x_LF, powerspectrum_f(x_LF), alpha=0.5, color="#51A6D8",
plt.gca().fill_between(x_HF, powerspectrum_f(x_HF), alpha=0.5, color="#ABF31F",

plt.gca().set_xlim(0, 0.75)
plt.gca().set_ylim(0)
plt.xlabel("Frequency (Hz)")
plt.ylabel("Density")

plt.legend()
plt.show()
```

FFT spectrum values display the absolute power (ms^2) of signal energy within component bands. In theory, the physiological correlates of these frequency band ranges are:

- **High Frequency (HF)** = Parasympathetic dominance
- **Low Frequency (LF)** = Sympathetic dominance
- **Very Low Frequency (VLF)** = Origins are not well defined — but low VLF has been associated with higher mortality

> *This seemingly ideal model representing autonomic nervous system balance (i.e. **HF:LF** ratio) is actually much more complex. On one hand, efferent vagal activity has been found as a major contributor to the **HF** component, but the interpretation of the **LF** component remains shrouded, and is continually up for debate in literature.*

Now lets use some code to spit us out a numeric summary of our Frequency-domain metrics

```
def freq_domain(fxx, pxx):

    #frequency bands: very low frequency (VLF), low frequency (LF), high frequen
    cond_VLF = (fxx >= 0) & (fxx < 0.04)
    cond_LF = (fxx >= 0.04) & (fxx < 0.15)
    cond_HF = (fxx >= 0.15) & (fxx < 0.4)

    #calculate power in each band by integrating the spectral density using trap
    VLF = trapz(pxx[cond_VLF], fxx[cond_VLF])
    LF = trapz(pxx[cond_LF], fxx[cond_LF])
    HF = trapz(pxx[cond_HF], fxx[cond_HF])

    #total power sum
    total_power = VLF + LF + HF

    # calculate power in each band by integrating the spectral density
    vlf = trapz(pxx[cond_VLF], fxx[cond_VLF])
    lf = trapz(pxx[cond_LF], fxx[cond_LF])
    hf = trapz(pxx[cond_HF], fxx[cond_HF])


    #peaks (Hz) in each band
    peak_VLF = fxx[cond_VLF][np.argmax(pxx[cond_VLF])]
    peak_LF = fxx[cond_LF][np.argmax(pxx[cond_LF])]
    peak_HF = fxx[cond_HF][np.argmax(pxx[cond_HF])]

    #fractions
    LF_nu = 100 * lf / (lf + hf)
    HF_nu = 100 * hf / (lf + hf)

    results = {}
    results['Power VLF (ms2)'] = VLF
    results['Power LF (ms2)'] = LF
    results['Power HF (ms2)'] = HF
    results['Power Total (ms2)'] = total_power

    results['LF/HF'] = (LF/HF)
    results['Peak VLF (Hz)'] = peak_VLF
    results['Peak LF (Hz)'] = peak_LF
    results['Peak HF (Hz)'] = peak_HF

    results['Fraction LF (nu)'] = LF_nu
    results['Fraction HF (nu)'] = HF_nu

    return results

freq_domain(fxx, pxx)
```

Conclusion

We learned how to take a raw ECG signal, extracted some meaningful features, and plotted lots. Everybody loves a good plot. We only covered the tip of the iceberg, there are much more advanced frequency based analyses, and many other non-linear and geometric features.

Most importantly, what will you do with these quantifications of the hearts performance? What can you inference about the physiological state of your autonomic nervous system based on HRV? Can you use these metrics to indicate stress or disease risk factor?

These are curious topics that I may cover in future posts!

Happy heart rate exploring!

. . .

https://github.com/kulasbart/ECG-processing_HRV/blob/master/HRV_ECG_analysis.ipynb

You can find my full GitHub repo along with the data I used here. There are also many public research data sets for heart data, you will find plenty on kaggle as well.

If you enjoyed the post, please consider following me on ***Medium*** 🙂

Ecg | Heart Rate Variability | Heart Rate | Python | Wearables

326 3

Written by Bartek Kulas

138 Followers

Data Science | AI/ML | Biotech https://www.linkedin.com/in/bartek-kulas-06896a1a2/

Follow

More from Bartek Kulas

Bartek Kulas in Better Programming

Independent Component Analysis

Bartek Kulas

Classifying Sentiment using Deep

(ICA) and Automated Component...

Independent Component Analysis (ICA) is a computational method used to decompose ...

6 min read · Jun 8, 2023

68

Learning: A BERT Project

Sentiment analysis, aka 'opinion mining', is a natural language processing (NLP) techniqu...

11 min read · Apr 19, 2023

59

See all from Bartek Kulas

Recommended from Medium

Shaw Talebi in Towards Data Science

The Wavelet Transform

An Introduction and Example

· 6 min read · Dec 21, 2020

600 5

Rehan Azhar

Extracting Time-Domain and Frequency-Domain Features from...

Introduction

3 min read · Oct 12, 2023

43 1

Lists

Coding & Development
11 stories · 505 saves

Predictive Modeling w/ Python
20 stories · 1007 saves

Practical Guides to Machine Learning
10 stories · 1206 saves

ChatGPT
21 stories · 521 saves

Tiya Vaj

1D signal vs 2 signal

Absolutely! Here's a breakdown of 1D and 2D signals, along with some visual examples to...

1 min read · Feb 17, 2024

Reyansh Bahl

EEG Signal Analysis With Python

Introduction

7 min read · Nov 6, 2023

8

Juan C Olamendy

Deep TDA. A new dimensionality reduction algorithm

Introduction

3 min read · Oct 19, 2023

S. Do.

Computing Mutual Information Matrix with Python.

These days I'm involved in a personal project that I find quite interesting. It involves...

9 min read · Dec 29, 2023

48 1 1

See more recommendations