```
def PDF_to_PNG(doc_ID):

    from PIL import Image

    pdf_path = filenames[doc_ID] + '.pdf'
    pdf_document = fitz.open(pdf_path)
    zoom = 2 # to increase the resolution
    mat = fitz.Matrix(zoom, zoom)

    for page_num in range(len(pdf_document)):
        page = pdf_document.load_page(page_num)
        pix = page.get_pixmap(matrix = mat) # or (dpi = 300)
        img = Image.frombytes("RGB", [pix.width, pix.height], pix.samples)
        filename = "PDF" + str(doc_ID) + "_" + str(page_num) + '.png'
        img.save(filename)
        print('Converting PDFs to Image ... ' + filename)

    return()

# set save_PNG = True to save PDF slides as PNG

save_PNG = False
if save_PNG:
    for doc_ID in (0, ):
        PDF_to_PNG(doc_ID)
```

1.6 Pattern detection and compression for electrocardiogram data

The goal is to automate the job of technicians who review electrocardiagram data (ECG) to identify various heart conditions, including predicting heart attacks as early as possible. In this section, I describe the first step towards building such an agent (to be integrated to an LLM system) with a focus on data compression, extracting heart beats in the data and classifying them based on detected patterns. The next step consists of labeling the clusters as benign or not, based on the patient heart rhythm history and other features (age, BMI and so on).

The data comes from the large public ECG repository Physionet.org, located here. The normalized sample `p04000_s00` analyzed here consists of about 1 million measurements with 250 values per second, thus lasting about an hour. A subset is pictured in Figure 1.14, with a smaller time period in Figure 1.15.

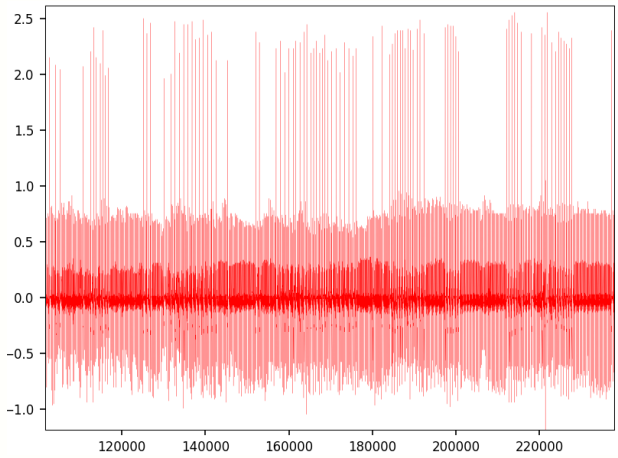

Figure 1.14: ECG data, about 10 minutes long

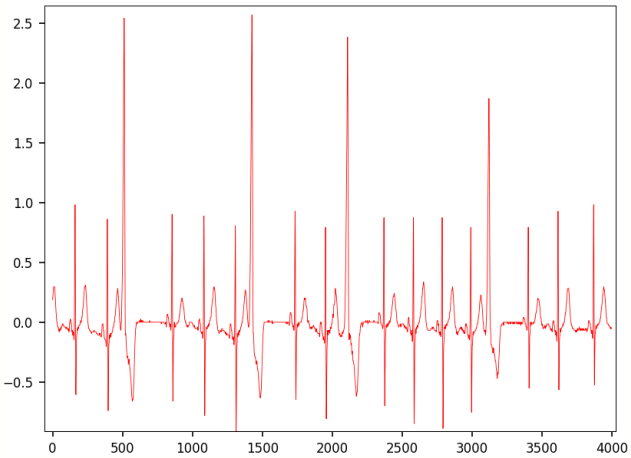

Figure 1.15: ECG data, about 16 seconds long

1.6.1 Compressing ECG signals

Typical compression rates for this type of data range from 3:1 to 4:1, with some algorithms doing a lot better depending on the signal. Here I show a simple, effective lossless compression technique achieving a 5.3 compression rate while the gzip ratio on the same data is 4.1. First, let ξ_t be the observed value at time t. Also, let $\Delta_t = \xi_{t+1} - \xi_t$ be the first order differences and $B_t = (\Delta_t, \Delta_{t+1}, \ldots, \Delta_{t+7})$. On the signal investigated here with 10^6 values, there are only 64,000 distinct combinations for B_0, B_8, B_{16} and so on, so you can store them in a dictionary with 2 bytes per combination. The size of the dictionary is 128KB. In the signal, replace the 125,000 consecutive B_0, B_8, B_{16} and so on by the corresponding 2-byte codes in the dictionary.

Now, after this operation, the size of the compressed signal is 378KB = 128KB + 250KB. The 128KB is for the dictionary storage, the 250KB for the encoded signal delta. Add ξ_0 at the beginning to allow for lossless reconstruction. The original file consisted of 10^6 values ξ_t, each taking 2 bytes, thus the size was 2MB. Note that my technique may not work for larger files, as the dictionary will quickly require more than 2 bytes per code as it grows. But you can slice your signal into pieces that are short enough and keep a separate 128KB dictionary for each slice.

1.6.2 Pattern detection and heart beat classification

I now discuss isolating the heart beats of variable duration visible in Figure 1.15 and classifying them according to their behavior. A standard method to detect heart beats is the Pan-Tompkins algorithm [Wiki]. In this section I describe a simpler technique. In the code in section 1.6.3, heart beats are referred to as `events`. My technique found between 4000 and 8000 of them in a 1-hour time period, depending on the thresholds in my model.

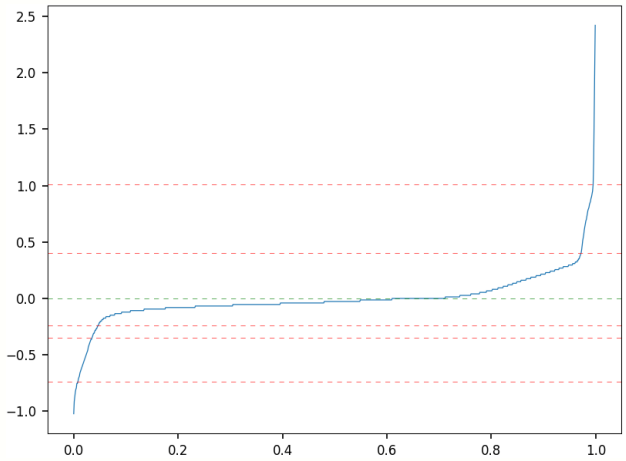

Figure 1.16: Empirical quantile function for signal ξ_t, with automatically detected thresholds (dashed lines)

Figure 1.17: Each dot is a heart beat. Duration on Y-axis, start time on X-axis. Color represents type.

Figure 1.16 shows the empirical quantile function for the values ξ_t. It is highly skewed, with visible change points where the curve exhibits an "elbow". These points are good indicators of the 4 nested bands present in Figure 1.14. Different techniques are available to detect them, for instance the elbow method [Wiki] or based on second derivatives with the gradient function in Numpy. I use the estimated curvature C_t of the quantile function Q_t. It is zero where Q_t is a flat line, and maximum where the curvature is strongest. See "threshold detection" in the code in section 1.6.3. The computation is as follows:

- Let $A_t = (t - \epsilon, Q_{t-\epsilon})$, $B_t = (t + \epsilon, Q_{t+\epsilon})$, and $P_t = (t, Q_t)$.
- Then C_t is the distance between P_t and the line segment $[A_t, B_t]$, that is,

$$C_t = \frac{1}{2} \cdot \frac{Q_{t-\epsilon} - 2Q_t + Q_{t+\epsilon}}{\sqrt{1 + (Q'_t)^2}} \quad \text{with } Q'_t = \frac{Q_{t+\epsilon} - Q_{t-\epsilon}}{2\epsilon}. \tag{1.9}$$

Once the change points in Q_t are found, it is easier to isolate heart beats with the corresponding thresholds. In the code, I use the `events` array to store the heart beats and the features attached to them: start time, width of first spike, maximum ξ_t, duration, minimum ξ_t, and number of values below baseline. See function `get_events`. Depending on the thresholds used, these "events" represent heart beats, parts of them, or a few consecutive heart beats lumped together into a single event, for instance in case the patient has missed or irregular heart beats.

Selected heart beat events with corresponding features are listed in Table 1.7. These events are ordered by start time. The total number is 7579 over a 4000–second time period, based on `threshold1` set to 0.25 in the code. By increasing the threshold to the next level 0.50, the number of events goes down to 4632, more in line with the average duration of a heart beat. Each second has 250 measurements (values). The "Lows" column counts the number of values below baseline ($\xi_t = 0$). The "Width" column indicates the spread of the first spike used as starting point for an event, measured as the number of consecutive values above the threshold. I now describe the "Label" column, resulting from clustering the heart beats events.

1.6.2.1 Heart beats clustering

I run unsupervised kmeans clustering on the 5 rightmost columns on the events table (see extract in Table 1.7) to group them into 8 categories. The "Label" column shows the category each event was assigned to. In Figure 1.17, each dot represents an event; the color represents its category. The X-axis represents the normalized start time while the Y-axis represents the number of values (duration) for each event. Here are the next steps:

- Do the same analysis for a large number of 1-hour datasets, both for the same patient and for other ones with known heart problems. In the process, increase the number of event categories.
- Assign a meaningful name to each category. Have an expert further categorize them as normal, abnormal but benign, alarming, or serious.
- For each patient, look at the distribution of event labels, both in absolute numbers and sequentially over time. The goal is to identify event sequences linked to heart attacks and other heart problems.

Finally, it is interesting to note the analogy between the heart beats in Figure 1.15, and the the generic synthetic signal in Figure 4.13 used to test a new type of deep neural networks that handles this type of signal quite well.

EventID	Label	Width	Max ξ_t	Duration	Min ξ_t	Lows
7546	5	6	0.417	8	0.188	0
7547	3	9	0.350	115	-0.161	44
7548	0	6	0.686	71	-0.861	40
7549	3	3	0.255	69	-0.121	35
7550	2	26	2.582	243	-0.807	82
7551	0	7	0.901	72	-0.699	11
7552	3	5	0.269	116	-0.148	23
7553	4	6	0.780	186	-0.686	101
7554	0	8	1.022	69	-0.538	21
7555	3	9	0.269	117	-0.161	67
7556	0	7	1.022	71	-0.699	28
7557	3	7	0.269	115	-0.175	77

Table 1.7: Selected heart beat events with corresponding features

1.6.3 Dataset and Python code

The source code named `dat_electrocardiogram.py` is also on my Google drive, here.

```
import wfdb
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import matplotlib.colors as mcolors

mpl.rcParams['axes.linewidth'] = 0.5
plt.rcParams['xtick.labelsize'] = 8
plt.rcParams['ytick.labelsize'] = 8
plt.rcParams['legend.fontsize'] = 'x-small'

# First, put the following files in your local directory to do little test:
#
# p04000_s00.hea
# p04000_s00.atr
# p04000_s00.dat
#
# Source: https://physionet.org/content/icentia11k-continuous-ecg/1.0/p04/p04000/#files-panel

record = wfdb.rdrecord('p04000_s00')

#--- Or to read data straight from Physionet website rather than locally as above
# record_name = '100'
# physionet_database_directory = 'mitdb'
# record = wfdb.rdrecord(record_name, pn_dir=physionet_database_directory)

#--- Info about record:
# record.p_signal: The signal data in physical units (if available).
# record.d_signal: The signal data in digital units.
# record.fs: The sampling frequency.
# record.sig_name: The names of the signals.
```

```
# record.units: The units of the signals.

#--- Other stuff you can do
# wfdb.plot_wfdb(record, title='Record p04000_s00 from Physionet Apnea ECG')
# for item in record.__dict__:
#    print(item)

signal = record.p_signal[:,0]
x_time = range(len(signal))
print("Signal lenth:", len(x_time))
print("Frequency:", record.fs) # value is 250/second

start = 10000
end = 990000

#--- Compute quantiles

arr_q = []
xval = []
incr = 0.001
for p in np.arange(0, 1, incr):
   q = np.quantile(signal[start:end], p)
   arr_q.append(q)
   xval.append(p)
plt.plot(xval, arr_q, linewidth=0.6)


#--- Thresholds detection

def dist_to_line(P, A, B):
   # distance between point P and line containing points A, B
   num = (P[0]-A[0])*(B[1]-A[1]) - (P[1]-A[1])*(B[0]-A[0])
   denum = np.sqrt((B[0]-A[0])**2 + (B[1]-A[1])**2)
   return(num/denum)

arr_thresh = np.zeros(len(arr_q))
offset = 2
for k in range(offset, len(arr_q)-offset):
   P = [xval[k], arr_q[k]]
   A = [xval[k-offset], arr_q[k-offset]]
   B = [xval[k+offset], arr_q[k+offset]]
   arr_thresh[k] = dist_to_line(P, A, B)

pary = 4
for k in range(len(arr_thresh)):
   thresh = arr_thresh[k]
   k1_min = max(0, k-pary)
   k1_max = min(len(arr_thresh)-1, k+pary)
   tmax = np.max(arr_thresh[k1_min:k1_max])
   k2_min = max(0, k-offset)
   k2_max = min(len(arr_thresh)-1, k+offset)
   dy = arr_q[k2_max] - arr_q[k2_min]
   if thresh == tmax and thresh > 0.0005 and dy > 0.04:
      plt.axhline(y=arr_q[k], color='r', linewidth = 0.3, dashes=[10, 10])
plt.axhline(y=0, color='g', linewidth = 0.3, dashes=[10, 10])
plt.show()


#---- Compressing the signal

def update_hash(hash, key, count):
   if key in hash:
      hash[key] += count
   else:
      hash[key] = 1
   return(hash)

arr_delta = signal[start:end] - signal[start-1:end-1]

hash_delta = {}
t = 0
block_size = 8
while t + block_size < len(arr_delta):
   vector = ()
   for idx in range(block_size):
      vector = (*vector, arr_delta[t+idx])
```

```
    update_hash(hash_delta, vector, 1)
    t += block_size

hash_delta = dict(sorted(hash_delta.items(), key=lambda item: item[1], reverse=True))
print("Number of unique blocks:", len(hash_delta))


#--- Detect heart beats (called "events")

def get_events(threshold, low):

    events = []
    t = start
    old_t_start = -1
    while t < end:
       if signal[t] > threshold:
          t_start = t
          width = 1
          max = - 1
          min = 9999.99
          lows = 0
          while t < end and signal[t] > threshold:
             if signal[t] > max:
                max = signal[t]
             width += 1
             t += 1
          while t < end and signal[t] <= threshold:
             if signal[t] < min:
                min = signal[t]
             if signal[t] < low:
                lows += 1
             t += 1
          duration = t - t_start
          plows = lows/duration
          events.append([t_start, width, max, duration, min, lows])
          old_t_start = t_start
       else:
          t += 1
    events = np.array(events)
    return(events)

threshold1 = 0.25
threshold2 = -0.05
events = get_events(threshold1, threshold2)


#--- Events clustering

from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

X = events[:, -5:]
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Generate 20 distinct colors (RGB tuples, values 0-1)
cmap = plt.get_cmap('tab20')
colors_rgb = [cmap(i) for i in range(20)]
events_color = []

ngroups = 8
if ngroups > 20:
   print("Not enough colors to show the categories")
   exit()
kmeans = KMeans(n_clusters=ngroups, random_state=0, n_init=10)
kmeans.fit(X_scaled)
labels = kmeans.labels_ ### predict(X)
centroids = kmeans.cluster_centers_

for k in range(len(labels)):
   label = labels[k]
   color = colors_rgb[label]
   events_color.append(color)
   print("%5d %3d %4d %5.3f %3d %5.3f %2d" %
         (k, label, events[k,1], events[k,2], events[k,3], events[k,4],events[k,5]))

```

```
184 plt.scatter(events[:,0], events[:, 3], s=2.6, c=events_color)
185 plt.show()
```