```
import copy
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from mediapipe import solutions
from mediapipe.framework.formats import landmark_pb2
import cv2
import mediapipe as mp
from mediapipe.tasks import python
from mediapipe.tasks.python import vision
from scipy.signal import butter, lfilter
from scipy.stats import pearsonr
from scipy.fftpack import dct, idct
from scipy.fft import fft, ifft
import time


# Used to visualize facial landmark detection results
# Obtained from model code examples provided by Google:
#
https://colab.research.google.com/github/googlesamples/mediapipe/blob/mai
n/examples/face_landmarker/python/%5BMediaPipe_Python_Tasks%5D_Face_Landm
arker.ipynb
def draw_landmarks_on_image(rgb_image, detection_result):
  face_landmarks_list = detection_result.face_landmarks
  annotated_image = np.copy(rgb_image)

  # Loop through the detected faces to visualize.
  for idx in range(len(face_landmarks_list)):
    face_landmarks = face_landmarks_list[idx]

    # Draw the face landmarks.
    face_landmarks_proto = landmark_pb2.NormalizedLandmarkList()
    face_landmarks_proto.landmark.extend([
      landmark_pb2.NormalizedLandmark(x=landmark.x, y=landmark.y,
z=landmark.z) for landmark in face_landmarks
    ])

    solutions.drawing_utils.draw_landmarks(
        image=annotated_image,
        landmark_list=face_landmarks_proto,
        connections=mp.solutions.face_mesh.FACEMESH_TESSELATION,
        landmark_drawing_spec=None,
        connection_drawing_spec=mp.solutions.drawing_styles
        .get_default_face_mesh_tesselation_style())
    solutions.drawing_utils.draw_landmarks(
        image=annotated_image,
        landmark_list=face_landmarks_proto,
        connections=mp.solutions.face_mesh.FACEMESH_CONTOURS,
        landmark_drawing_spec=None,
        connection_drawing_spec=mp.solutions.drawing_styles
        .get_default_face_mesh_contours_style())
    solutions.drawing_utils.draw_landmarks(
        image=annotated_image,
        landmark_list=face_landmarks_proto,
        connections=mp.solutions.face_mesh.FACEMESH_IRISES,
          landmark_drawing_spec=None,
          connection_drawing_spec=mp.solutions.drawing_styles
```

```
          .get_default_face_mesh_iris_connections_style())

  return annotated_image


# Separates input video into frames
def extract_frames(video_path):
    frames = []

    # Open the video file
    cap = cv2.VideoCapture(video_path)
    fps = -1

    if not cap.isOpened():
        print(f"Error: Cannot open video file {video_path}")
        return [], fps

    while True:
        # Read a frame from the video
        ret, frame = cap.read()

        if not ret:
            break

        frames.append(frame)

    fps = cap.get(cv2.CAP_PROP_FPS)

    cap.release()
    return frames, fps

# Applies butterworth bandpass filter on each landmark, over time
def apply_filter_to_array(data, lowcut, highcut, fs, order=5):
    # Apply the moving average filter to each feature point (column)
    return np.apply_along_axis(lambda x: butter_bandpass_filter(x,
lowcut, highcut, fs, order), axis=0, arr=data)

def butter_bandpass_filter(data, lowcut, highcut, fs, order=5):
    b, a = butter_bandpass(lowcut, highcut, fs, order=order)
    y = lfilter(b, a, data)
    return y

def butter_bandpass(lowcut, highcut, fs, order=5):
    return butter(order, [lowcut, highcut], fs=fs, btype='band')

# Applies moving average filter to each landmark, over time
def apply_moving_average_filter(data, window_size):
    # Apply the moving average filter to each feature point (column)
    return np.apply_along_axis(lambda x: moving_average_filter(x,
window_size), axis=0, arr=data)

# To make sure the moving average filter is applied properly, the
beginning and end portions of the signal
# corresponding to half the moving average window size will be discarded
def moving_average_filter(signal, window_size):
    half_w = window_size // 2
    T = len(signal)
```

```
    filtered_signal = np.zeros_like(signal, dtype=float)
    for t in range(half_w, T - half_w):
        filtered_signal[t] = np.mean(signal[t - half_w : t + half_w + 1])

    return filtered_signal[half_w : T - half_w]

# Produces heart rate estimation from filtered signal
def extract_heart_rate(data, fs):
    # Step 1: Apply PCA to the data
    principal_components, load_matrix = perform_pca(data)

    # Step 2: Apply DCT to the components and find the most periodic one
    most_periodic_component =
apply_dct_and_find_heartbeat_component(principal_components)

    # Step 3: Estimate the heart rate using FFT on the inverse DCT of the
periodic component
    heart_rate = estimate_heart_rate(most_periodic_component, fs)

    return heart_rate, most_periodic_component

def perform_pca(data):
    # Subtract the mean to center the data
    data_centered = data - np.mean(data, axis=0)

    # Compute the covariance matrix of the data
    covariance_matrix = np.cov(data_centered, rowvar=False)

    # Eigenvalue decomposition
    eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)

    # Sort eigenvectors by eigenvalues in descending order
    sorted_indices = np.argsort(eigenvalues)[::-1]
    eigenvectors_sorted = eigenvectors[:, sorted_indices]

    # Project the data onto the eigenvectors
    principal_components = np.dot(data_centered, eigenvectors_sorted)

    return principal_components, eigenvectors_sorted


# Apply DCT and select the most periodic component
def apply_dct_and_find_heartbeat_component(principal_components):
    # Apply DCT to each component (along the time axis)
    dct_components = np.apply_along_axis(dct, axis=0,
arr=principal_components)

    # Find the component with the highest frequency magnitude (most
periodic)
    frequencies_magnitude = np.abs(dct_components)
    most_periodic_component_index =
np.argmax(np.sum(frequencies_magnitude, axis=0))  # Column-wise max

    # Extract the most periodic component
    most_periodic_component = dct_components[:,
most_periodic_component_index]

    return most_periodic_component
```

```
# Estimate HR from the most periodic component
def estimate_heart_rate(periodic_component, fs):
    # Inverse DCT to recover the time-domain signal
    time_domain_signal = idct(periodic_component, type=2, norm='ortho')

    # Apply FFT to the recovered signal
    fft_signal = fft(time_domain_signal)

    # Compute the frequencies corresponding to the FFT bins
    freqs = np.fft.fftfreq(len(time_domain_signal), d=1 / fs)
    # Get the magnitude of the FFT
    fft_magnitude = np.abs(fft_signal)

    # Find the frequency corresponding to the first harmonic
    peak_freq_index = np.argmax(fft_magnitude[1:]) + 1
    peak_freq = freqs[peak_freq_index]

    # Convert the frequency to heart rate (in beats per minute)
    heart_rate = peak_freq * 60

    return heart_rate

if __name__ == '__main__':
    # Iterate over participants in the tested activity
    activity = "05"
    # Comment for activity 4: participant 2 is missing for activity 4
    participants_in_activity = ["00", "01", "02", "03", "04", "05", "06",
"07", "08", "09", "10", "11", "12", "13", "14", "15", "16"]
    # Uncomment for activity 4
    # participants_in_activity = ["00", "01", "03", "04", "05", "06",
"07", "08", "09", "10", "11", "12", "13", "14", "15", "16"]

    # Data Measured
    mae_all = []
    sde_all = []
    rmse_all = []
    pearson_corr_all = []

    mae_less_20 = []
    sde_less_20 = []
    rmse_less_20 = []
    pearson_corr_less_20 = []

    mae_stretched_one_frame = []
    sde_stretched_one_frame = []
    rmse_stretched_one_frame = []

    landmark_detection_time_all = []
    landmark_selection_and_filtering_time_all = []
    extracting_hr_from_filtered_signal_time_all = []

    frames_discarded_percentage_all = []
    frames_discarded_less_20 = []
    frames_discarded_percentage_passed = []
    frames_discarded_percentage_failed = []
```

```
    all_frames_dropped = []

    # Hyper parameters
    min_confidence = 0.75
    size_of_moving_average_window = 5  # in seconds
    window_duration = 20  # in seconds
    between_windows_interval = 10  # in seconds

    # https://storage.googleapis.com/mediapipe-
assets/documentation/mediapipe_face_landmark_fullsize.png
    # ROI landmarks
    left_eye_landmarks = [70, 63, 105, 66, 107, 55, 193, 122, 188, 114,
47, 100, 101, 118, 117, 111, 143, 156, 119, 120,
                          121, 128, 245, 244, 112, 26, 22, 23, 24, 110,
25, 130, 247, 30, 29, 27, 28, 56, 190, 243, 35,
                          124, 46, 53, 52, 65, 233, 232, 231, 230, 229,
228, 31, 226, 113, 225, 224, 223, 222, 221, 189]
    right_eye_landmarks = [336, 296, 334, 293, 300, 383, 372, 340, 346,
347, 348, 349, 350, 357, 465, 464, 417, 285,
                           295, 282, 283, 276, 353, 265, 446, 261, 448,
449, 450, 451, 452, 453, 341, 463, 414, 413,
                           441, 442, 443, 444, 445, 342, 467, 359, 255,
339, 254, 253, 252, 256, 286, 258, 257, 259,
                           260, ]
    forehead_landmarks = [103, 104, 67, 69, 109, 108, 9, 151, 10, 338,
337, 297, 299, 332, 333]
    lower_face_landmarks = [36, 301, 50, 187, 205, 206, 203, 207, 216,
186, 92, 165, 167, 164, 393, 391, 322, 410, 287,
                            436, 426, 423, 266, 425, 280, 411, 427]
    ROI_landmarks = left_eye_landmarks + right_eye_landmarks +
forehead_landmarks + lower_face_landmarks

    # Stable landmarks
    left_eye_socket_and_pupil = [159, 158, 157, 173, 133, 155, 154, 153,
145, 144, 163, 7, 33, 246, 161, 160, 468]
    right_eye_socket_an_pupil = [384, 385, 386, 387, 388, 466, 263, 249,
390, 373, 374, 380, 381, 382, 362, 369, 398,
                                 473]
    vertical_line_through_nose = [8, 168, 6, 197, 195, 5, 4, 1, 19]
    mouth_outline = [146, 91, 181, 84, 17, 314, 405, 321, 375, 409, 270,
269, 267, 0, 37, 39, 40, 185]
    stable_landmarks = left_eye_socket_and_pupil +
right_eye_socket_an_pupil + vertical_line_through_nose + mouth_outline

    selected_landmarks = ROI_landmarks + stable_landmarks
    selected_landmarks = np.asarray(selected_landmarks)

    # Warning for presence of duplicate landmarks
    # if (len(np.unique(selected_landmarks)) != len(selected_landmarks)):
    #     print("Duplicate landmarks present!")
    #
    #     for landmark in selected_landmarks:
    #         if selected_landmarks.tolist().count(landmark) > 1:
    #             print(landmark, " is duplicate")

    # Create a FaceLandmarker object.
    base_options =
python.BaseOptions(model_asset_path='face_landmarker.task')
```

```
    # Face quality assessment implemented through model confidence values
    options = vision.FaceLandmarkerOptions(base_options=base_options,
                                           output_face_blendshapes=False,
output_facial_transformation_matrixes=True,
                                           num_faces=1,
min_face_detection_confidence=min_confidence,
min_face_presence_confidence=min_confidence,
min_tracking_confidence=min_confidence)
    detector = vision.FaceLandmarker.create_from_options(options)

    less_20_dropped = True
    stretched_one_frame = False

    for participant in participants_in_activity:
        less_20_dropped = True
        stretched_one_frame = False

        # Break video down into images
        video_path = f"D:\\{participant}\\{activity}\c920-1.avi"  # Path
to the input video file
        frames, video_fps = extract_frames(video_path)  # Call the
function to extract frames

        landmark_z_positions_over_time = []
        frame_ids = []

        start = time.time()
        # Analyze each frame one by one and store landmark z positions
        for idx, frame in enumerate(frames):
            # Convert BGR to RGB
            rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

            # Create mediapipe image from frame
            image = mp.Image(image_format=mp.ImageFormat.SRGB,
data=rgb_frame)

            # Detect face landmarks
            detection_result = detector.detect(image)

            # Uncomment line below to visualize face landmarks detected
by model
            # annotated_image =
draw_landmarks_on_image(image.numpy_view(), detection_result)

            if len(detection_result.face_landmarks) > 0:
                landmarks = detection_result.face_landmarks[0]
                landmark_z_positions_frame = []
                for landmark in landmarks:
                    landmark_z_positions_frame.append(landmark.z)


landmark_z_positions_over_time.append(landmark_z_positions_frame)
                frame_ids.append(idx)
```

```
            # Uncomment section below to visualize face landmarks
detected by model - press any key to view next frame
            # cv2.imshow(f"Frame {idx}", cv2.cvtColor(annotated_image,
cv2.COLOR_RGB2BGR))
            # if cv2.waitKey(0) & 0xFF == ord('q'):  # Press 'q' to quit
viewing frames early
            #     break
            # cv2.destroyAllWindows()

        frame_times = []
        for frame_id in frame_ids:
            frame_times.append(frame_id / video_fps)

        frames_discarded = len(frames) -
len(landmark_z_positions_over_time)
        fraction_dropped = frames_discarded * 1.0 / len(frames)
        frames_discarded_percentage_all.append(fraction_dropped)

        # If all frames were dropped, then it is impossible to evaluate
the video
        if frames_discarded >= (len(frames) -
size_of_moving_average_window * video_fps):
            all_frames_dropped.append(participant)
            frames_discarded_percentage_failed.append(fraction_dropped)
            continue

        # If more than 20% of frames were discarded skip to next video
        if (fraction_dropped > 0.2):
            # print(f"video from participant {participant} was skipped.
Fraction of frames dropped by FQA: {fraction_dropped}")
            less_20_dropped = False
            # continue
        if less_20_dropped:
            frames_discarded_less_20.append(fraction_dropped)

        end_detection = time.time()
        landmark_detection_time_all.append((end_detection - start) *
1000)

        # Select specific landmarks:
        selected_landmark_z_positions = []

        for frame in landmark_z_positions_over_time:
            landmarks = []
            for landmark in selected_landmarks:
                landmarks.append(frame[landmark])
            selected_landmark_z_positions.append(landmarks)

        # Subtract initial position to only keep in the relative movement
over time
        selected_landmark_z_positions_over_time_movement =
copy.deepcopy(selected_landmark_z_positions)

        for frame in selected_landmark_z_positions_over_time_movement:
            for i in range(len(frame)):
                frame[i] -= selected_landmark_z_positions[0][i]
```

```
        selected_landmark_z_positions_over_time_movement =
np.array(selected_landmark_z_positions_over_time_movement)

        # Uncomment section below for graph of first landmark movement
before filtering
        # plt.figure(figsize=(10, 6))
        # plt.plot(frame_times,
selected_landmark_z_positions_over_time_movement[:, 0],
        #          color='b')
        # plt.xlabel('Time (seconds)', fontsize=18)
        # plt.ylabel('Z-axis movement (arbitrary unit)', fontsize=18)
        # plt.title(f'Movement of first landmark over time, before any
filtering\n [activity {activity}, participant {participant}]',
fontsize=21)
        # plt.legend()
        # plt.xticks(fontsize=16)
        # plt.yticks(fontsize=16)
        # plt.grid(True, linestyle='--', alpha=0.6)
        # plt.show()

        # 8-th order Butterworth filter
        filtered =
apply_filter_to_array(selected_landmark_z_positions_over_time_movement,
0.75, 5.0, video_fps, order=8)

        # Uncomment section below for graph of first landmark movement
after butterworth bandpass filter
        # plt.figure(figsize=(10, 6))
        # plt.plot(frame_times, filtered[:, 0], color='b')
        # plt.xlabel('Time (seconds)', fontsize=18)
        # plt.ylabel('Z-axis movement (arbitrary unit)', fontsize=18)
        # plt.title(f'Movement of first landmark over time, after
butterworth filter\n [activity {activity}, participant {participant}]',
fontsize=21)
        # plt.legend()
        # plt.xticks(fontsize=16)
        # plt.yticks(fontsize=16)
        # plt.grid(True, linestyle='--', alpha=0.6)
        # plt.show()

        # Moving average filter
        size_of_moving_window = int(size_of_moving_average_window *
video_fps)
        moving_average_filtered = apply_moving_average_filter(filtered,
size_of_moving_window)
        frame_times = frame_times[size_of_moving_window // 2:-
size_of_moving_window // 2]

        # Uncomment section below for graph of first landmark movement
after butterworth bandpass filter and moving average filter
        # plt.figure(figsize=(13, 6))
        # plt.plot(frame_times, moving_average_filtered[:, 0], color='b')
        # plt.xlabel('Time (seconds)', fontsize=18)
        # plt.ylabel('Z-axis movement (arbitrary unit)', fontsize=18)
        # plt.title(f'Movement of first landmark over time, after
butterworth filter\n and moving average filter [activity {activity},
participant {participant}]', fontsize=21)
        # plt.legend()
```

```
        # plt.xticks(fontsize=16)
        # plt.yticks(fontsize=16)
        # plt.grid(True, linestyle='--', alpha=0.6)
        # plt.show()

        end_filtering = time.time()
        landmark_selection_and_filtering_time_all.append((end_filtering -
end_detection) * 1000)

        # Break up video using a sliding window
        hr_measurements = []
        window_start_times = []
        window_end_times = []
        window_duration_frames = int(window_duration * video_fps)
        last_starting_point = len(moving_average_filtered) -
int(window_duration * video_fps)
        step_in_frames = int(video_fps * between_windows_interval)

        for i in range(0, last_starting_point, step_in_frames):
            end_frame = i + window_duration_frames
            hr, periodic_comp =
extract_heart_rate(moving_average_filtered[i:end_frame], video_fps)
            hr_measurements.append(hr)
            window_start_times.append(frame_times[i])
            window_end_times.append(frame_times[end_frame])

        end_extraction = time.time()

extracting_hr_from_filtered_signal_time_all.append((end_extraction -
end_filtering) * 1000)

        # Get HR measurement from ECG
        df = pd.read_csv(f"D:\\{participant}\\{activity}\\viatom-
raw.csv") # Path to ECG data csv file

        ecg_hr = df.loc[df[' ECG HR'] < 0, ' ECG HR'] = df.loc[df[' ECG
HR'] < 0, ' ECG HR'].abs()
        milliseconds = df['milliseconds']

        milliseconds_normalized = milliseconds - milliseconds.iloc[0]

        hr_measurements_ecg = []
        for i in range(0, last_starting_point, step_in_frames):
            start_frame = i
            end_frame = i + window_duration_frames

            start_time = frame_times[start_frame] * 1000
            end_time = frame_times[end_frame] * 1000

            # Filter rows where milliseconds are between start_time and
end_time
            readings_in_window = df[(milliseconds_normalized >=
start_time) & (milliseconds_normalized <= end_time)]
            # If there are any readings in the window, calculate the
average of ECG HR
            avg_ecg_hr = readings_in_window[' ECG HR'].mean()
            hr_measurements_ecg.append(avg_ecg_hr)
```

```
        # If frames are discarded leaving less than one window, then
whatever data is left available is used for one estimation
        if len(hr_measurements) == 0:
            stretched_one_frame = True
            start_frame = 0
            end_frame = len(moving_average_filtered) - 1

            start_time = frame_times[start_frame] * 1000
            end_time = frame_times[end_frame] * 1000

            hr, periodic_comp =
extract_heart_rate(moving_average_filtered[start_frame:end_frame],
video_fps)
            hr_measurements.append(hr)
            window_start_times.append(frame_times[start_frame])
            window_end_times.append(frame_times[end_frame])

            # Filter rows where milliseconds are between start_time and
end_time
            readings_in_window = df[(milliseconds_normalized >=
start_time) & (milliseconds_normalized <= end_time)]
            # If there are any readings in the window, calculate the
average of ECG HR
            avg_ecg_hr = readings_in_window[' ECG HR'].mean()
            hr_measurements_ecg.append(avg_ecg_hr)

        window_mid_times = (np.array(window_start_times) +
np.array(window_end_times)) / 2
        x_labels = [f"{start:.2f}-{end:.2f}" for start, end in
zip(window_start_times, window_end_times)]

        plt.figure(figsize=(10, 6))
        plt.plot(window_mid_times, hr_measurements, label='Proposed
Method Heart Rate Estimation',
                         color='b')
        plt.plot(window_mid_times, hr_measurements_ecg, label="Heart Rate
ECG", color="r")
        plt.xlabel('Window Intervals (seconds)', fontsize=18)
        plt.ylabel('Heart Rate (bpm)', fontsize=18)
        plt.title(f'HR estimation for {window_duration} second window,
sliding every {between_windows_interval} seconds\n [activity {activity},
participant {participant}]', fontsize=21)
        plt.xticks(ticks=window_mid_times, labels=x_labels, fontsize=16)
        plt.yticks(fontsize=16)
        plt.legend(fontsize=16)
        plt.grid(True, linestyle='--', alpha=0.6)
        plt.show()

        hr_measurements_ecg = np.array(hr_measurements_ecg)
        hr_measurements = np.array(hr_measurements)

        if stretched_one_frame == True:
            frames_discarded_percentage_failed.append(fraction_dropped)

            error = hr_measurements_ecg - hr_measurements
            # 1. Mean Absolute Error (MAE)
            mae = np.mean(np.abs(error))
            mae_all.append(mae)
```

```
            mae_stretched_one_frame.append(mae)

            # 2. Standard Deviation of Error (SDE)
            sde = np.std(error)
            sde_all.append(sde)
            sde_stretched_one_frame.append(sde)

            # 3. Root Mean Squared Error (RMSE)
            rmse = np.sqrt(np.mean(error ** 2))
            rmse_all.append(rmse)
            rmse_stretched_one_frame.append(rmse)

        else:
            frames_discarded_percentage_passed.append(fraction_dropped)

            # Statistical measures
            error = hr_measurements_ecg - hr_measurements
            # 1. Mean Absolute Error (MAE)
            mae = np.mean(np.abs(error))
            mae_all.append(mae)
            if less_20_dropped:
                mae_less_20.append(mae)

            # 2. Standard Deviation of Error (SDE)
            sde = np.std(error)
            sde_all.append(sde)
            if less_20_dropped:
                sde_less_20.append(sde)

            # 3. Root Mean Squared Error (RMSE)
            rmse = np.sqrt(np.mean(error**2))
            rmse_all.append(rmse)
            if less_20_dropped:
                rmse_less_20.append(rmse)

            # 4. Pearson Correlation Coefficient
            if len(hr_measurements) >= 2:
                pearson_corr, _ = pearsonr(hr_measurements_ecg,
hr_measurements)
                if not np.isnan(pearson_corr):
                    pearson_corr_all.append(pearson_corr)
                    if less_20_dropped:
                        pearson_corr_less_20.append(pearson_corr)

    print(f"No frame was identified for the following videos:
{all_frames_dropped}")

    avg_mae = np.mean(mae_all)
    avg_sde = np.mean(sde_all)
    avg_rmse = np.mean(rmse_all)
    avg_pearson_corr = np.mean(pearson_corr_all)

    print(f"Average MAE: {avg_mae}")
    print(f"Average SDE: {avg_sde}")
    print(f"Average RMSE: {avg_rmse}")
    print(f"Average Pearson Correlation: {avg_pearson_corr}")

    avg_mae_less_20 = np.mean(mae_less_20)
```

```
    avg_sde_less_20 = np.mean(sde_less_20)
    avg_rmse_less_20 = np.mean(rmse_less_20)
    avg_pearson_corr_less_20 = np.mean(pearson_corr_less_20)

    print(f"Average Less 20 MAE: {avg_mae_less_20}")
    print(f"Average Less 20 SDE: {avg_sde_less_20}")
    print(f"Average Less 20 RMSE: {avg_rmse_less_20}")
    print(f"Average Less 20 Pearson Correlation:
{avg_pearson_corr_less_20}")

    avg_mae_stretched = np.mean(mae_stretched_one_frame)
    avg_sde_stretched = np.mean(sde_stretched_one_frame)
    avg_rmse_stretched = np.mean(rmse_stretched_one_frame)

    print(f"Average MAE One-window-stretched: {avg_mae_stretched}")
    print(f"Average SDE One-window-stretched: {avg_sde_stretched}")
    print(f"Average RMSE One-window-stretched: {avg_rmse_stretched}")

    print(f"Average Less 20 MAE: {avg_mae_less_20}")
    print(f"Average Less 20 SDE: {avg_sde_less_20}")
    print(f"Average Less 20 RMSE: {avg_rmse_less_20}")
    print(f"Average Less 20 Pearson Correlation:
{avg_pearson_corr_less_20}")

    avg_landmark_extraction_time = np.mean(landmark_detection_time_all)
    avg_filtering_time =
np.mean(landmark_selection_and_filtering_time_all)
    avg_hr_extraction_time =
np.mean(extracting_hr_from_filtered_signal_time_all)

    print(f"Average Landmark extraction time:
{avg_landmark_extraction_time}")
    print(f"Average filtering time: {avg_filtering_time}")
    print(f"Average hr extraction time: {avg_hr_extraction_time}")

    avg_frames_discarded_percentage =
np.mean(frames_discarded_percentage_all)
    avg_frames_discarded_percentage_less_20 =
np.mean(frames_discarded_less_20)
    avg_frames_discarded_percentage_passed =
np.mean(frames_discarded_percentage_passed)
    avg_frames_discarded_percentage_failed =
np.mean(frames_discarded_percentage_failed)

    print(f"Average percentage of frames discarded:
{avg_frames_discarded_percentage}")
    print(f"Average percentage of frames discarded (<=20% frames
dropped): {avg_frames_discarded_percentage_less_20}")
    print(f"Average percentage of frames discarded in passed scenarios:
{avg_frames_discarded_percentage_passed}")
    print(f"Average percentage of frames discarded in failed scenarios:
{avg_frames_discarded_percentage_failed}")
```