ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»
Факультет компьютерных наук
Образовательная программа «Программная Инженерия»

Подп. и дата	
Инв. № дубл.	
Взам. инв. №	
Подп. и дата	
Инв. № подл	

СОГЛАСОВАНО
Руководитель направления,
Лаборатория Искусственного
Интеллекта, Сбербанк
_____/ К. С. Егоров /
«___»_____2021 г.

УТВЕРЖДАЮ
Академический руководитель
образовательной программы
«Программная инженерия», канд. техн.
наук, профессор ДПИ ФКН
_____В. В. Шилов
«___»_____2021 г.

АВТОМАТИЧЕСКАЯ РАСШИФРОВКА ЭКГ ПО ХОЛТЕРУ

Текст программы

ЛИСТ УТВЕРЖДЕНИЯ

RU.17701729.04.01-01 12 01-1 ЛУ

Исполнитель
Студент группы БПИ197
_____/ П. О. Кулешова /
«___»_____ 2021 г.

Москва 2021

УТВЕРЖДЕН

RU.17701729.04.01-01 12 01-1

Подп. и дата	
Инв. № дубл.	
Взам. инв. №	
Подп. и дата	
Инв. № подл	

АВТОМАТИЧЕСКАЯ РАСШИФРОВКА ЭКГ ПО ХОЛТЕРУ

Текст программы

RU.17701729.04.01-01 12 01-1

Листов 24

Москва 2021

ОГЛАВЛЕНИЕ

1. ТЕКСТ ПРОГРАММ

1.1.data_utils.py

```
import numpy as np
import pandas as pd


from glob import glob
import os

ECG_PATH = '/large/datasets/holter/ecg'
RR2_PATH = '/large/datasets/holter/rr2'
VOTED_PATH = '/large/datasets/holter/voted'

ECG_FILE = '/large/datasets/holter/train_ecg_v2.npy'
ANN_FILE = '/large/datasets/holter/train_ann_v2.npy'

def get_patients():
    ecgs = sorted(list(glob(f'{ECG_PATH}/*.ecg')))
    patients = list()
    for ecg_path in ecgs:
        patient = ecg_path.split('/')[-1].split('.')[0]
        if os.path.isfile(f'{RR2_PATH}/{patient}.rr2'):
            patients.append(int(patient))
    return patients

def get_train():
    ecgs = sorted(list(glob(f'{ECG_PATH}/*.ecg')))
    patients = list()
    test = get_test()
    for ecg_path in ecgs:
        patient = int(ecg_path.split('/')[-1].split('.')[0])
        if os.path.isfile(f'{RR2_PATH}/{patient}.rr2'):
            if patient not in test:
                patients.append(patient)
    return patients


def get_test():
    ecgs = sorted(list(glob(f'{ECG_PATH}/*.ecg')))
    patients = list()
    for ecg_path in ecgs:
```

```
            patient = int(ecg_path.split('/')[-1].split('.')[0])
            if os.path.isfile(f'{VOTED_PATH}/{patient}.rr2'):
                patients.append(patient)
        return patients


    def get_ecg(patient, path=None):
        path = path if path else ECG_PATH
        dt = np.dtype('uint16')
        dt = dt.newbyteorder('>')
        with open(f'{path}/{patient}.ecg', 'rb') as f:
            signal = np.frombuffer(f.read(), dtype=dt)
        signal = signal[:(signal.shape[0] // 3) * 3]
        signal = signal.reshape(-1, 3).T
        signal = signal[:2, :].astype('uint16')

        return signal



    def get_ann(patient, path=None):
        path = path if path else RR2_PATH
        with open(f'{path}/{patient}.rr2', 'rb') as f:
            signal = np.frombuffer(f.read(), dtype=np.int32)
        result = decode_ann(signal)
        return result


    def decode_ann(signal, fix=True):
        signal = signal.reshape((-1, 30)).copy()
        result = pd.DataFrame(signal)
        cols = ['nom', 'pos_p1', 'pos_q1', 'pos_R1', 'pos_s1', 'pos_t11', 'pos_t12',
'ampl1', 'ampl1_znak',
                'd_extrem1', 'ST1', 'ST1_znak', 'pos_p2', 'pos_q2', 'pos_R2', 'pos_s2',
'pos_t21', 'pos_t22',
                'ampl2', 'ampl2_znak', 'd_extrem2', 'ST2', 'ST2_znak', 'tip_qrs',
'nom_frm', 'p_art', 'shum',
                'extr_pauz', 'epizod', 'fp']
        result.columns = cols

        if fix:
            tip_qrs_map = {-2: 3, -1: 4, 0: 5, 1: 0, 2: 1, 3: 2, 4: 3}
```

```
            result['tip_qrs'] = result['tip_qrs'].map(tip_qrs_map)

            extr_pauz_map = {0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 10: 0}
            result['extr_pauz'] = result['extr_pauz'].map(extr_pauz_map)

        result['tip_qrs'] = result['tip_qrs'].astype('int32')
        result['extr_pauz'] = result['extr_pauz'].astype('int32')
        result['is_pauz'] = result['extr_pauz'].isin([2, 3, 4]).astype('int32')

        return result
```

1.2.models.py

```
import pytorch_modules_2

import torch
from sklearn.model_selection import train_test_split
from tqdm import tqdm
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


class AutoEncoder:
    def __init__(self,
                 channels_mult=1,
                 bottleneck=16,
                 batch_norm=True,
                 valid_size=0.3,
                 seed=None,
                 aug_params=None,
                 num_workers=0,
                 batch_size=32,
                 device='cpu',
                 lr=0.003,
                 num_epochs=10,
                 log_file='log.csv',
                 weights_file='weights.pt',
                 train_net=True,
                 preload=None):

        self.valid_size = valid_size
        self.seed = seed
        self.aug_params = aug_params
        self.num_workers = num_workers
        self.batch_size = batch_size
        self.device = device
        self.lr = lr
        self.num_epochs = num_epochs
        self.log_file = log_file
        self.weights_file = weights_file
        self.preload = preload
        self.train_net = train_net

        self.net = pytorch_modules_2.ECGAutoEncoder(channels_mult=channels_mult,
                                                    bottleneck=bottleneck,
                                                    batch_norm=batch_norm)
        self.criterion = torch.nn.MSELoss()
```

```
        self.optimizer = torch.optim.Adam(self.net.parameters(), lr=self.lr,
weight_decay=0.0)

    def fit(self, ecgs):

        if self.preload is not None:
            self.net = torch.load(self.preload)

        if self.train_net:

            train, valid = train_test_split(ecgs, test_size=self.valid_size,
random_state=self.seed)

            train_ds = PytorchDS(train, aug_params=self.aug_params)
            valid_ds = PytorchDS(valid, aug_params=None)

            train_dl = torch.utils.data.DataLoader(train_ds,
                                                   batch_size=self.batch_size,
                                                   num_workers=self.num_workers,
                                                   shuffle=True)

            valid_dl = torch.utils.data.DataLoader(valid_ds,
                                                   batch_size=self.batch_size,
                                                   num_workers=self.num_workers,
                                                   shuffle=True)

            log = list()
            best_loss = 1e9
            for epoch in range(self.num_epochs):
                train_log = self.run_epoch(train_dl, train=True)
                with torch.no_grad():
                    valid_log = self.run_epoch(valid_dl, train=False)

                row = dict()
                row['epoch'] = epoch
                row.update({f'train_{key}': val for key, val in train_log.items()})
                row.update({f'valid_{key}': val for key, val in valid_log.items()})
                log.append(row)
                pd.DataFrame(log).to_csv(self.log_file)

                print(pd.DataFrame(log))
                if row['valid_loss'] < best_loss:
                    best_loss = row['valid_loss']
                    torch.save(self.net, self.weights_file)

        self.net = torch.load(self.weights_file)

    def transform(self, ecgs):
        self.net.cpu()
        self.net.eval()
        bottlenecks = list()
        ecgs = torch.from_numpy(ecgs).view(-1, 2, 128).float()
        steps = int(np.ceil(ecgs.shape[0] / self.batch_size))
        with torch.no_grad():
            for step in range(steps):
                start = step * self.batch_size
                end = start + self.batch_size
                out = self.net(ecgs[start: end])

                bottlenecks.append(out['bottleneck'].detach().cpu().numpy()[:, :, 0])
        return np.concatenate(bottlenecks)
```

```
    def run_epoch(self, dl, train=True):
        if train:
            self.net.train()
        else:
            self.net.eval()

        self.net.to(self.device)

        epoch_loss = 0
        plotted = False
        for i, batch in tqdm(enumerate(dl), total=len(dl)):
            ecg_raw = batch['ecg_raw'].to(self.device)
            ecg_aug = batch['ecg_aug'].to(self.device)

            out = self.net(ecg_aug)
            loss = self.criterion(out['autoencoded'][:, :, 16:-16], ecg_raw[:, :, 16:-
16])
            epoch_loss += loss.item()
            if train:
                self.optimizer.zero_grad()
                loss.backward()
                self.optimizer.step()

            if not plotted and ecg_raw.shape[0] >= 3:
                plotted = True
                plt.figure(figsize=(20, 10))
                for i in range(3):
                    ecg_in_raw = ecg_raw.cpu().numpy()[i]
                    ecg_in_aug = ecg_aug.cpu().numpy()[i]
                    ecg_out = out['autoencoded'].detach().cpu().numpy()[i]

                    plt.subplot(3, 2, 2 * i + 1)
                    plt.plot(ecg_in_raw[0])
                    plt.plot(ecg_out[0])
                    plt.plot(ecg_in_aug[0], alpha=0.5)
                    plt.ylim(-10, 10)

                    plt.subplot(3, 2, 2 * i + 2)
                    plt.plot(ecg_in_raw[1])
                    plt.plot(ecg_out[1])
                    plt.plot(ecg_in_aug[1], alpha=0.5)
                    plt.ylim(-10, 10)
                plt.show()

        epoch_loss /= len(dl)

        return {'loss': epoch_loss}


class PytorchDS(torch.utils.data.Dataset):
    def __init__(self, ecgs, aug_params=None):
        self.ecgs = ecgs
        self.aug_params = aug_params

    def __len__(self):
        return self.ecgs.shape[0]

    def __getitem__(self, index):
        ecg = self.ecgs[index].reshape((2, -1)).astype('float32')
        result = dict()
        result['ecg_raw'] = ecg
        result['ecg_aug'] = self.augment(ecg.copy())
```

```
        return result

    def augment(self, ecg):
        if self.aug_params is not None:
            if np.random.random() < self.aug_params['zero_chan_p']:
                chan = np.random.choice([0, 1])
                ecg[chan] = 2 * (np.random.random(ecg[chan].shape) - 0.5) *
self.aug_params['zero_chan_noise']
            ecg += 2 * (np.random.random(ecg.shape) - 0.5) * self.aug_params['noise']
        return ecg
```

1.3. pipeline_tools.py

```
import functools
from time import time
import torch
import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.metrics import homogeneity_score

import data_utils, processing



def valid_split(df, valid_size, seed=None):
    patients = sorted(df['patient'].unique())
    train, valid = train_test_split(patients, test_size=valid_size,
random_state=seed)
    train = df[df['patient'].isin(train)]
    valid = df[df['patient'].isin(valid)]
    return train, valid

@functools.lru_cache(maxsize=1)
def get_data(anns_path, ecgs_path):
    anns = pd.read_csv(anns_path)
    ecgs = list()
    for x in range(5425):
        ecgs.append(np.load(f"{ecgs_path}/data{x}.npy"))
    ecgs=np.concatenate(ecgs)
    return anns, ecgs



def experiment_v2(params):
```

```
print('Loading data ... ', end='')
start = time()

anns = np.load(params['anns_file'])
anns = pd.DataFrame(anns)
anns.columns = ['pos_R1', 'tip_qrs', 'patient']
anns['index'] = range(anns.shape[0])

ecgs = np.load(params['ecgs_file'])
print(f'Done in {int(time() - start)} sec.')
print(f'Anns shape: {anns.shape}')
print(f'ECGs shape: {ecgs.shape}')
print()

print('Splitting train/validation ... ', end='')
start = time()

train = anns
train_ecgs = ecgs[train['index'].values]

train_ecgs = train_ecgs.reshape(train_ecgs.shape[0], -1)
print(f'Done in {int(time() - start)} sec.')
print('Train ECGs shape:', train_ecgs.shape)
print()

print('Training embedder ... ', end='')
start_time = time()
embedder = params['embedder_class'](**params['embedder_params'])
embedder.fit(train_ecgs)

window = params['window']
halfwindow = window // 2
result = list()
test_patients = data_utils.get_test()
print('Test patients:', len(test_patients))
for patient in test_patients:

    print('Patient', patient)
    patient_start = time()
    print('Loading data ... ', end='')
    start_time = time()
    ann = data_utils.get_ann(patient, path=data_utils.VOTED_PATH)
```

```
            ann = ann[ann['tip_qrs'].isin([0, 2])]
            ann = ann[ann['shum'] == 0]
            if (ann['tip_qrs'] == 2).mean() > 0.1:
                ecg = data_utils.get_ecg(patient)
                end_time = time()
                print(f'Done in {int(end_time - start_time)} seconds.')

                print('Processing ecg ... ', end='')
                start_time = time()
                ecg = processing.process_ecg(ecg).astype('float32')
                ecgs = list()
                goods = list()
                for index, row in ann.iterrows():
                    ecg_slice = processing.get_slice(ecg, row['pos_R1'], window,
maxval=20, threshold=10)
                    if ecg_slice is not None:
                        ecgs.append(ecg_slice)
                        goods.append(True)
                    else:
                        goods.append(False)

                if np.any(goods):
                    ecgs = np.stack(ecgs)

                    ann['goods'] = goods
                    end_time = time()
                    print(f'Done in {int(end_time - start_time)} seconds.')

                    print('Training embedder ... ', end='')
                    start_time = time()
                    embedder.fit(train_ecgs)
                    end_time = time()
                    print(f'Done in {int(end_time - start_time)} seconds.')

                    print('Embeddings ... ', end='')
                    start_time = time()
                    embeddings = embedder.transform(ecgs)
                    embeddingd_mean = embeddings.mean(axis=0)[None, :]
                    embeddings_std = embeddings.std(axis=0)[None, :]
                    embeddings -= embeddingd_mean
                    embeddings /= embeddings_std
                    end_time = time()
```

```
                print(f'Done in {int(end_time - start_time)} seconds.')

                print('Clustering ... ', end='')
                start_time = time()
                clusterer = params['clusterer_class'](**params['clusterer_params'])
                clusterer.fit(embeddings)
                labels = clusterer.labels_
                ann['label'] = -1
                ann.loc[ann['goods'], 'label'] = labels
                end_time = time()
                print(f'Done in {int(end_time - start_time)} seconds.')

                n_clusters = np.unique(labels).shape[0] - 1
                homogeneity = homogeneity_score(ann['tip_qrs'], ann['label'])
                noise = (labels == -1).sum() / labels.shape[0]

                row = dict()
                row['patient'] = patient
                row['n_qrs'] = ann.shape[0]
                row['tip_qrs_2_fraq'] = (ann['tip_qrs'] == 2).mean()
                row['n_clusters'] = n_clusters
                row['homogeneity'] = homogeneity
                row['noise'] = noise
                result.append(row)
                pd.DataFrame(result).to_csv(f'logs/pipe_logs_{params["name"]}.csv')
                pd.DataFrame(ann).to_csv(f'logs/ann_{patient}.csv')
                patient_end = time()
                print(f'Patietn done in {(patient_end - patient_start)} seconds')
            else:
                print('Done. Bad ecg')

        else:
            print('Done. Low Wide QRS count')

    return n_clusters, homogeneity, noise

def run(params):
    embedder = params['embedder_class'](**params['embedder_params'])
    embedder.net = torch.load(embedder.weights_file)
    ann = data_utils.get_ann(params['patient'], path=params["RR2_path"])
    ecg = data_utils.get_ecg(params['patient'], path=params["ECG_path"])
    ecg = processing.process_ecg(ecg).astype('float32')
```

```
        ecgs = list()
        goods = list()
        for index, row in ann.iterrows():
            ecg_slice = processing.get_slice(ecg, row['pos_R1'], params['window'],
maxval=20, threshold=10)
            if ecg_slice is not None:
                ecgs.append(ecg_slice)
                goods.append(True)
            else:
                goods.append(False)

        if np.any(goods):
            ecgs = np.stack(ecgs)
            ann['goods'] = goods
            embeddings = embedder.transform(ecgs)
            clusterer = params['clusterer_class'](**params['clusterer_params'])
            clusterer.fit(embeddings)
            labels = clusterer.labels_
            ann['label'] = -1
            for x in range(embeddings.shape[1]):
                ann[f'emb_{x}']=0
                ann.loc[ann['goods'], f'emb_{x}'] = embeddings[:, x]
            ann.loc[ann['goods'], 'label'] = labels
            ann.to_csv(f"{params['output_path']}/anns_{params['patient']}.csv")
        else:
            print('Done. Bad ecg')
```

1.4.processing.py

```
    import scipy.signal
    import numpy as np
    import numba


    def filter_signal(ts, rate, low_freq=None, high_freq=None, order=4):
        if low_freq:
            lb_n_freq = low_freq / (rate/2)
            b, a = scipy.signal.butter(order, lb_n_freq, 'high')
            ts = scipy.signal.filtfilt(b, a, ts)

        if high_freq:
            hb_n_freq = high_freq / (rate/2)
            b, a = scipy.signal.butter(order, hb_n_freq, 'low')
```

```
            ts = scipy.signal.filtfilt(b, a, ts)

        ts = ts.copy()
        return ts



    def process_ecg(ecg, trend=100, downsample=2, low_freq=None, high_freq=40, clip=20,
divide_by=50):
        processed = ecg.astype('float32')
        for i in range(2):
            s = processed[i]
            mask = s != 0
            s[:100] = np.median(s[mask][:1000])
            s[-100:] = np.median(s[mask][-1000:])
            s = interp_holes(s)

            s -= np.median(s)
            kernel = np.ones(trend) / trend
            mean = np.convolve(s, kernel, mode='same')
            mean = np.convolve(mean, kernel, mode='same')
            s -= mean
            s = filter_signal(s, 250, low_freq=low_freq, high_freq=high_freq)
            s /= divide_by
            s = np.clip(s, -clip, +clip)
            processed[i] = s

        processed = processed[:2, :].astype('float16')
        processed = processed[:, ::downsample]
        return processed



    @numba.njit
    def interp_holes(arr):
        zeros = np.where(arr == 0)[0]
        if len(zeros) == 0:
            return arr

        first = zeros[0]
        last = zeros[0]
        for zero in zeros[1:]:
```

```
            if zero - last == 1:
                last = zero
            else:
                left = arr[first - 1]
                right = arr[last + 1]
                arr[first - 1: last + 2] = np.linspace(left, right, last - first + 3)
                first = zero
                last = zero

        left = arr[first - 1]
        right = arr[last + 1]
        arr[first - 1: last + 2] = np.linspace(left, right, last - first + 3)

        return arr


    def get_slice(ecg, center, window, maxval=20, threshold=10):
        start = center - window // 2
        if start < 0:
            start = 0
            end = window
        else:
            end = start + window

        if end > ecg.shape[1]:
            end = ecg.shape[1]
            start = end - window

        assert start >=0 and end <= ecg.shape[1], f'Start: {start}, End: {end}, Shape:
{ecg.shape[1]}'

        ecg_slice = ecg[:, start: end]
        if ((ecg_slice == -maxval) | (ecg_slice == maxval)).sum() < threshold:
            ecg_slice -= np.median(ecg_slice, axis=1)[:, None]
            std = ecg_slice.std(axis=1)[:, None]
            std[std == 0] = 1e-6
            ecg_slice /= std
            ecg_slice = np.clip(ecg_slice, -maxval, maxval)
            return ecg_slice

        else:
            return None
```

1.5. pytorch_moduls_2.py

```
import torch
from torch import nn
from torch.nn import functional as F
import numpy as np

class Conv1dBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, pooling=True,
batch_norm=False):
        super().__init__()
        self.pooling = pooling
        self.batch_norm = batch_norm
        self.conv = nn.Conv1d(in_channels, out_channels, kernel_size,
padding=kernel_size//2, bias=True, stride=1)
        self.act = nn.ReLU()
        if self.pooling:
            self.maxpool = nn.MaxPool1d(2)
        if self.batch_norm:
            self.bn = nn.BatchNorm1d(out_channels)

    def forward(self, x):
        x = self.conv(x)
        if self.batch_norm:
            x = self.bn(x)
        x = self.act(x)
        if self.pooling:
            x = self.maxpool(x)
        return x

class DownBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size):
        super().__init__()
        self.conv1 = Conv1dBlock(in_channels, out_channels, kernel_size,
pooling=False)
        self.conv2 = Conv1dBlock(out_channels, out_channels, kernel_size,
pooling=True)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
```

```
            return x


    class UpBlock(nn.Module):
        def __init__(self, in_channels, out_channels, kernel_size):
            super().__init__()
            self.conv1 = Conv1dBlock(in_channels, out_channels, kernel_size,
pooling=False)
            self.conv2 = Conv1dBlock(out_channels, out_channels, kernel_size,
pooling=False)

        def forward(self, x):
            x = F.interpolate(x, scale_factor=2, mode='linear', align_corners=False)
            x = self.conv1(x)
            x = self.conv2(x)
            return x




    class ECGEncoder(nn.Module):
        def __init__(self, channels, bottleneck=16, batch_norm=True):
            super().__init__()

            self.conv1 = Conv1dBlock(          2, channels[0], 7, pooling=False,
batch_norm=batch_norm)
            self.conv2 = Conv1dBlock(channels[0], channels[1], 7, pooling=False,
batch_norm=False)

            self.down1 = DownBlock(channels[1], channels[2], 5)
            self.down2 = DownBlock(channels[2], channels[3], 5)
            self.down3 = DownBlock(channels[3], channels[4], 5)
            self.down4 = DownBlock(channels[4], channels[5], 3)
            self.down5 = DownBlock(channels[5], channels[6], 3)

            self.conv3 = Conv1dBlock(channels[6], channels[6], 3, pooling=False,
batch_norm=batch_norm)
            self.conv4 = Conv1dBlock(channels[6], channels[6], 3, pooling=False,
batch_norm=False)
```

```
            self.bottleneck = nn.Conv1d(channels[6], bottleneck, 4)
            self.bottlenect_act = nn.Tanh()

        def forward(self, x):
            x = self.conv1(x)
            x = self.conv2(x)

            x = self.down1(x)
            x = self.down2(x)
            x = self.down3(x)
            x = self.down4(x)
            x = self.down5(x)

            x = self.conv3(x)
            x = self.conv4(x)

            x = self.bottleneck(x)
            x = self.bottlenect_act(x)
            return x


    class ECGDecoder(nn.Module):
        def __init__(self, channels, bottleneck=16, batch_norm=True):
            super().__init__()

            self.bottleneck = nn.Conv1d(bottleneck, channels[6], 1)
            self.bottlenect_act = nn.ReLU()

            self.up1 = UpBlock(channels[6], channels[6], 3)
            self.up2 = UpBlock(channels[6], channels[5], 3)
            self.up3 = UpBlock(channels[5], channels[4], 3)

            self.conv5 = Conv1dBlock(channels[4], channels[4], 3, pooling=False,
batch_norm=batch_norm)
            self.conv6 = Conv1dBlock(channels[4], channels[4], 3, pooling=False,
batch_norm=False)

            self.up4 = UpBlock(channels[4], channels[3], 5)
            self.up5 = UpBlock(channels[3], channels[2], 5)

            self.conv7 = Conv1dBlock(channels[2], channels[2], 3, pooling=False,
batch_norm=batch_norm)
```

```
            self.conv8 = Conv1dBlock(channels[2], channels[2], 3, pooling=False,
batch_norm=False)

            self.up6 = UpBlock(channels[2], channels[2], 7)
            self.up7 = UpBlock(channels[2], channels[2], 7)

            self.conv9 = Conv1dBlock(channels[2], channels[1], 3, pooling=False,
batch_norm=batch_norm)
            self.conv10 = Conv1dBlock(channels[1], channels[0], 3, pooling=False,
batch_norm=False)

            self.output = nn.Conv1d(channels[0], 2, 1)


        def forward(self, x):
            x = self.bottleneck(x)
            x = self.bottlenect_act(x)

            x = self.up1(x)
            x = self.up2(x)
            x = self.up3(x)

            x = self.conv5(x)
            x = self.conv6(x)

            x = self.up4(x)
            x = self.up5(x)

            x = self.conv7(x)
            x = self.conv8(x)

            x = self.up6(x)
            x = self.up7(x)

            x = self.conv9(x)
            x = self.conv10(x)

            x = self.output(x)
            return x
```

```
class ECGAutoEncoder(nn.Module):
    def __init__(self, channels_mult=1, bottleneck=16, batch_norm=True):
        super().__init__()

        channels = (16, 24, 32, 40, 48, 56, 64)
        channels = (np.array(channels) * channels_mult).astype('int32')

        self.encoder = ECGEncoder(channels, bottleneck=bottleneck,
batch_norm=batch_norm)
        self.decoder = ECGDecoder(channels, bottleneck=bottleneck,
batch_norm=batch_norm)

    def forward(self, x):
        embedding = self.encoder(x)
        output = self.decoder(embedding)

        result = dict()
        result['bottleneck'] = embedding
        result['autoencoded'] = output
        return result
```

1.6. train_advanced.py

```
import hdbscan
import sys
import os
import models, pipeline_tools, data_utils, processing
import numpy as np
import pandas as pd
from time import time

VALID_SIZE = 0.2
SEED = 333
COLUMN = 'tip_qrs'
ECG_FILE = 'train_ecg_my.npy'
ANN_FILE = 'train_ann_my.npy'
DOWNSAMPLE = 2
WINDOW = 128
NUM_WORKERS = 6


def process_patient(patient):
```

```
        ecgs = list()
        anns = list()

        ann = data_utils.get_ann(patient)
        ann = ann[ann['shum'] == 0]
        ann = ann[['pos_R1', COLUMN]]
        ann['patient'] = patient
        ann['pos_R1'] //= DOWNSAMPLE

        ecg = data_utils.get_ecg(patient)
        ecg = processing.process_ecg(ecg, downsample=DOWNSAMPLE)

        for index, row in ann.iterrows():
            ecg_slice = processing.get_slice(ecg, row['pos_R1'], WINDOW, maxval=20,
threshold=10)
            if ecg_slice is not None:
                ecgs.append(ecg_slice)
                anns.append(row)

        if len(anns) > 0:
            anns = pd.DataFrame(anns)
            ecgs = np.stack(ecgs)
        else:
            anns = pd.DataFrame([])
            ecgs = np.zeros((0, 2, WINDOW), dtype='float16')
        return anns, ecgs


    def expirement(ecg_file, ann_file, patient):
        sys.path.append('../holter')
        params = dict()
        params['anns_file'] = ann_file
        params['ecgs_file'] = ecg_file
        params['valid_size'] = 0.3
        params['seed'] = 333
        params['embedder_class'] = models.AutoEncoder
        params['clusterer_class'] = hdbscan.HDBSCAN
        params['window'] = 128
        params['patient'] = patient

        embedder_params = dict()
        embedder_params['channels_mult'] = 1
```

```
        embedder_params['bottleneck'] = 32
        embedder_params['batch_norm'] = True
        embedder_params['valid_size'] = 0.3
        embedder_params['seed'] = 333
        embedder_params['aug_params'] = {'noise': 1.0, 'zero_chan_p': 0.5,
'zero_chan_noise': 2.0}
        embedder_params['num_workers'] = 0
        embedder_params['batch_size'] = 512
        embedder_params['device'] = 'cuda:0'
        embedder_params['lr'] = 0.003
        embedder_params['num_epochs'] = 5
        embedder_params['log_file'] = 'logs.csv'
        embedder_params['preload'] = 'weights_12.pt'
        embedder_params['weights_file'] = 'weights_12.pt'
        embedder_params['train_net'] = True

        clusterer_params = dict()
        clusterer_params['algorithm'] = 'boruvka_kdtree'
        clusterer_params['min_cluster_size'] = 100
        clusterer_params['cluster_selection_epsilon'] = 0.01
        clusterer_params['min_samples'] = 10

        params['embedder_params'] = embedder_params
        params['clusterer_params'] = clusterer_params
        pipeline_tools.run(params)


    def expirement_run(ecg_path, rr2_path, output_path, patient=None,):
        sys.path.append('../holter')
        params = dict()
        params['valid_size'] = 0.3
        params['seed'] = 333
        params['embedder_class'] = models.AutoEncoder
        params['clusterer_class'] = hdbscan.HDBSCAN
        params['window'] = 128
        params['patient'] = patient
        params["ECG_path"] = ecg_path
        params["RR2_path"] = rr2_path
        params["output_path"] = output_path
        embedder_params = dict()
        embedder_params['channels_mult'] = 1
        embedder_params['bottleneck'] = 32
```

```
        embedder_params['batch_norm'] = True
        embedder_params['valid_size'] = 0.3
        embedder_params['seed'] = 333
        embedder_params['aug_params'] = {'noise': 1.0, 'zero_chan_p': 0.5,
'zero_chan_noise': 2.0}
        embedder_params['num_workers'] = 0
        embedder_params['batch_size'] = 512
        embedder_params['device'] = 'cuda:0'
        embedder_params['lr'] = 0.003
        embedder_params['num_epochs'] = 5
        embedder_params['log_file'] = 'logs.csv'
        embedder_params['preload'] = 'weights_12.pt'
        embedder_params['weights_file'] = 'weights_12.pt'
        embedder_params['train_net'] = False

        clusterer_params = dict()
        clusterer_params['algorithm'] = 'boruvka_kdtree'
        clusterer_params['min_cluster_size'] = 100
        clusterer_params['cluster_selection_epsilon'] = 0.01
        clusterer_params['min_samples'] = 10

        params['embedder_params'] = embedder_params
        params['clusterer_params'] = clusterer_params
        pipeline_tools.run(params)


    def main():
        # path_ecg=f"/large/datasets/holter/ecg/{patient}.ecg"
        # path_rr2= f"/large/datasets/holter/rr2/{patient}.rr2"
        print("Введите путь до ЭКГ: \n")
        path_ecg = input()
        filename, file_extension = os.path.splitext(path_ecg)
        while not os.path.isfile(path_ecg) or file_extension != ".ecg":
            print("Некорректный путь до ЭКГ. Введите ещё раз:")
            path_ecg = input()
            filename, file_extension = os.path.splitext(path_ecg)
            # path_ecg=f"/large/datasets/holter/ecg/5050.ecg"
            # path_rr2 = f"/large/datasets/holter/rr2/5050.rr2"
        print("Введите путь до разметки:\n")
        path_rr2 = input()
        filename, file_extension = os.path.splitext(path_rr2)
        while not os.path.isfile(path_ecg) or file_extension != ".rr2":
```

```
            print("Некорректный путь до разметки. Введите ещё раз:")
            path_rr2 = input()
            filename, file_extension = os.path.splitext(path_rr2)
        print("Введите путь до выходного файла формата .csv:")
        path_output = input()
        filename, file_extension = os.path.splitext(path_output)
        arr = path_output.split("/")
        # /large/home/polindromka/project/Automatic-Interpretation-Of-Holter-
ECG/final/Holter/result.csv
        path="/".join(arr[:-1])
        while not os.path.isdir(path) or file_extension != ".csv":
            print("Некорректный путь до файла. Введите ещё раз:")
            path_output = input()
            filename, file_extension = os.path.splitext(path_output)
            arr = path_output.split("/")
            path = "/".join(arr[:-1])
        start = time()
        expirement_run(path_ecg, path_rr2, path_output)
        print(f'Done in {int(time() - start)} sec.')


    if __name__ == "__main__":
        main()
```

ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ

Изм.	Номера листов (страниц)				Всего листов (страниц) в документе	№ документа	Входящий № сопроводительного документа и дата	Подпись	Дата
	измененных	замененных	новых	аннулированных					