

Team: Anastasiia Stefanska

<b>How to run the code</b>	<b>1</b>
<b>1.1 Dataset Exploration</b>	<b>1</b>
<b>1.2. Modelling and Tuning</b>	<b>2</b>
<b>1.3 Data Augmentation and Feature Engineering</b>	<b>3</b>
1.3.1 Manual: HeartPy and augmentation with std	3
1.3.2 Snowflake classifier	4
<b>1.4 Data Reduction</b>	<b>4</b>
<b>2 Appendix - what did not work</b>	<b>5</b>

How to run the code

Python version 3.13.1, generated by python3 --version.

The code is in the Jupyter Notebook.

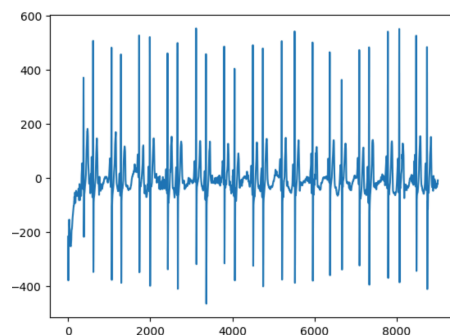
Requirements are in requirements.txt, generated by pip3 freeze. Please note requirements include packages for approaches which have been excluded in the end due to their performance or complexity to comprehend. Those are listed in the appendix.

Random seed in my example code is always 42, [reference to approach](#).

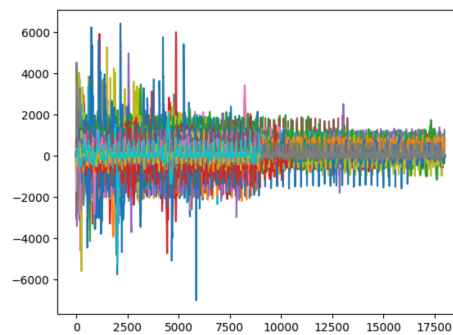
## 1.1 Dataset Exploration

I use basic statistics on the raw signal: count, min, max, difference between min and max, length, mean, median, standard deviation, variance, skew, kurtosis. I use each on row level, as these will be part of my features for classification. I also group these metrics by mean and std to see the class statistics and homogeneity. This helps to predict if a feature is going to work for classification.

I also plot random data and overall data profile.



First rows in training set



First 301 rows in training set

Observations:

1. From the plotting, data is noisier at the beginning.
2. Classes distribution is very not balanced, count per class: 0: 3638, 1: 549, 2: 1765, 3: 227. Class 0 dominant, rest - underrepresented.
3. Class 0 and 2 are very similar, class 3 is very noisy, class 1 has a smaller amplitude.
4. Signal is variable length, I will need to take this into account while building models.
5. Some statistics are different for class, but due to high std they are not likely to be useful on row level predictions (e.g. mean).
6. Class 3 looks to be uniformly distributed noise (med ~0, skew ~0), while the rest have signal asymmetries. I deduce the shape of the signal will be important.

7. I do not find a way to distinguish between class 0 and 2 by these statistics, I suspect this is going to be a problematic part.

Since I plan to use a fixed length signal for simplicity of further use (including splitting the data), I trim the signal to TARGET\_LENGTH at this step.

For the initial split I am going to use sklearn `train_test_split` with `test_size=0.2` and `stratify=y` to prevent losing my rare classes in validation set. Example split count of rows result by class, preserving balance proportion between classes:

Training:

val\_y = 0: count = 2910; val\_y = 1: count = 439; val\_y = 2: count = 1412; val\_y = 3: count = 182

Validation:

val\_y = 0: count = 728; val\_y = 1: count = 110; val\_y = 2: count = 353; val\_y = 3: count = 45

## 1.2. Modelling and Tuning

I want to test how well the models will work on already selected “basic statistics” features.

I am exploring the following models with the data I have generated:

- RandomForestClassifier
- LGBMClassifier
- LogisticRegression
- MLPClassifier
- SVC

At this moment I am using all models on generic features. My main metric of evaluation is weighted F1, as I know the imbalance of classes is high.

First I tried 3 random forests on different columns of data:

- Random forest on the fixed raw signal. F1 - 0.48.
- Random forest on the fixed raw signal and features. F1 - 0.47.
- Random forest on features only. **F1 - 0.56**. The winner from the 3, which I tuned.

Now I look on features only, and train and tune 4 more models to compare:

- LGBM classifier: F1 - 0.53.
- MLPClassifier: F1 - 0.45.
- LogisticRegression: F1 - 0.46.
- SVC with rbf kernel for non-linearity: F1 - 0.52.

Random forest still comes back as the winner, so I am going to use it in part 3.

What I think happens here is that every tuned model is in reality overtuned for class 1 as it is a very dominant class. I hope we can change that in part 3.

For tuning, I use GridSearchCV with the following options & results:

- RandomForestClassifier

```
'n_estimators': [100, 200], #total estimators
```

```
'max_depth': [None, 10, 20], #tree depth
```

```
'min_samples_split': [2, 5], #split if the node is minimum this size
```

```
'min_samples_leaf': [1, 2] #can be a leaf with this amount of examples
```

```
Best params {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
```

- LGBMClassifier

```
'n_estimators': [100, 200],
```

```
'learning_rate': [0.01, 0.1],
```

```
'max_depth': [5, 10, -1], # -1 - no limit
```

```
'num_leaves': [15, 31, 63],
```

```
'min_child_samples': [10, 20]
```

```
Best params {'learning_rate': 0.1, 'max_depth': 5, 'min_child_samples': 10, 'n_estimators': 200, 'num_leaves': 63}
```

- **MLPClassifier**

```
'hidden_layer_sizes': [(50,), (100,), (100, 50)], #layers structure  
'activation': ['relu', 'tanh'], #non-linear function for neurons  
'solver': ['adam', 'sgd'], #weight update strategy  
'learning_rate': ['constant', 'adaptive'],  
'alpha': [0.0001, 0.001] #weight decay
```

```
Best params {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (100, 50), 'learning_rate': 'constant', 'solver': 'adam'}
```

- **LogisticRegression**

```
'logreg__C': [0.01, 0.1, 1.0, 10.0],  
'logreg__penalty': ['l2', 'l1'],  
'logreg__solver': ['liblinear', 'saga']
```

```
Best params {'logreg__C': 10.0, 'logreg__penalty': 'l2', 'logreg__solver': 'saga'}
```

- **SVC**

```
'svc__C': [0.1, 1, 10], #regularization parameter  
'svc__gamma': ['scale', 0.01, 0.1, 1], #controls curvature  
'svc__kernel': ['rbf']
```

```
Best params {'svc__C': 10, 'svc__gamma': 'scale', 'svc__kernel': 'rbf'}
```

One thing which did not work for me was to change the verbosity of GridSearchCV to show the F1 score during tuning. I always just get the final params and F1 (see example below), but never the full picture of training. I think it might be a bug in verbosity.

## 1.3 Data Augmentation and Feature Engineering

### 1.3.1 Manual: HeartPy and augmentation with std

First, I use HeartPy and extract all the features it gives. For the noisy signals, it won't always be possible to extract, then I am going to just fill the row with dummy values (nan -> -99) and create a special column to signal that the value was noisy. Here are the columns I append to the data:

```
'bpm', 'ibi', 'sdnn', 'sdsd', 'rmssd', 'pnn20', 'pnn50', 'hr_max', 'sd1',  
'sd2', 's', 'sd1/sd2', 'breathingrate', 'vlf', 'lf', 'hf', 'lf/hf',  
'p_total', 'vlf_perc', 'lf_perc', 'hf_perc', 'lf_nu', 'hf_nu'
```

Additionally, I add a couple of metrics which I think could be interesting for ECG data:

- **normalized\_peak\_count** - frequency of repeated signal, normalized by length
- **max\_peak\_height** - strength of repeated signal, normalized by length
- **bpm\_variability** - irregularity of beat rate (std)
- **dominant\_freq** - the most powerful frequency, uses fast Fourier transform

After that, I standardize all data on column level and split it.

I do split before augmenting data to avoid spilling data between train and validation.

After the split, I only use train subset of data to generate new “noisy” signals similar to the ones already in the dataset. I use std which is different for different classes and is higher for the “noisy” class. After augmentation, I get:

Original training size: 4943

Augmented training size: 11640

Finally, I use the newly augmented data to train the model again, this time only focusing on random forest on features. **Weighted F1 is 0.70**. So despite deluding dominance of class 0, we are now having an overall better performance.

I am also curious to see which features rank as important, here is the final ranking on model on augmented data (val stands for importance in feature\_importances\_):

Name	Val	Name	Val	Name	Val	Name	Val
std_val	437	bpm	325	pnn50	259	bpm_variability	201
var_val	435	sdnn	302	autocorr_max_peak_height	245	sd1/sd2	196
rmssd	427	p_total	291	vlf_perc	244	med_val	184
lf	342	s	290	breathingrate	243	dominant_freq_hz	168
sd1	334	len_val	288	hr_max	233	heart_rate_bpm	163
sdsd	332	lf_nu	285	lf/hf	232	hf_nu	140
autocorr_norm_peak_count	332	hf	282	min_val	228	hf_perc	136
pnn20	329	vlf	276	max_val	217		
ibi	328	is_noisy	267	sd2	217		
diff_val	325	lf_perc	261	mean_val	204		

Interestingly, the top 10 features are a combination of initially selected statistics, HeartPy features and manually mined ECG-related features.

Upon using this model on the text data, unfortunately I do not get any sample predicted as class 0. Either this is a rather unconventional test data where the distribution of the classes is significantly different from train data, or my model is broken. To validate my assumption, I proceed to step 1.3.2.

### 1.3.2 Snowflake classifier

Now, since I suspect my model is broken, but I won't give up, I want to test some industry classifier on this data. I am going to feed my original and augmented data to Snowflake classifier and see how weighted F1 is behaving.

I use [SNOWFLAKE.ML.CLASSIFICATION](#) on features only. By design I am not allowed to tune anything on the data. I first test original features data, and get weighted F1 of 0.59, which looks promising (my own model was 0.56, so even if it overfits to class 0, it does it well). Upon creating a classification model in Snowflake on augmented data, I see a weighted F1 score of **0.81**.



I use the test data and get the same predictions as with my model: the class 0 is also missing in predictions on the test data. I deduce my own model is not broken and the test data is different distributed to train data. I attach my snowflake code for augmented data in snowflake.sql for optional reference. (Also seemingly good weighted F1, but since I cannot finetune it at all, I cannot use this as base for homework.)

### 1.4 Data Reduction



As mentioned in the very beginning, as I decided to work on the features of the data, I trimmed all the original data of warriors length to 9000 points. So my first technique is naive data trimming which achieves 0.7 weighted F1 on augmented data, and on Snowflake it is 0.8 weighted F1.

To further reduce the data, I had the following idea: only show every second or third value - I think 300 Hz is too much, 150 or 100 would be enough. To compare the data volume, I save original data and reduced data as csv.

Every third point:

 data_out_full.zip	Today, 20:14	79,9 MB	ZIP archive
 data_out.zip	Today, 19:51	30 MB	ZIP archive

Every second point:

 data_out_full.zip	Today, 20:14	79,9 MB	ZIP archive
 data_out_one_of_2.zip	Today, 21:44	43,6 MB	ZIP archive

After converting the df to a new (every second or third datapoint), I rerun the notebook.

One thing I need to adjust is TARGET\_LENGTH to 4500 or 3000 and sample\_rate to 150 or 100, as I halved or thirded the data.

Basic statistics (min, max etc) do not change significantly. Heart related statistics change, but I assume it preserves differences in properties.

1/3 data: weighted F1 on raw data features is 0.56. But I am losing weighted F1 from 0.70 to 0.65.

1/2 data: weighted F1 on raw data features is 0.56. But I am losing weighted F1 from 0.70 to 0.68.

In general, I only used naive methods, due to personal time constraints.

## 2 Appendix - what did not work

What I also tried and did not work:

- using window function to split data for 3000 items with overlap of 1500 - creates data splitting
- binary classifier for one class of features and ensemble of all - overfits
- train neural network on Colab T4 (free version) - see below.

Neural network #1:

```
def build_model(input_shape, n_classes, l2_reg, dropout_rate):
    model = Sequential([
        # Feature extraction CNN - repeated to get more features
        Conv1D(32, kernel_size=5, activation='relu', padding='same',
              kernel_regularizer=l2(l2_reg), input_shape=input_shape),
        BatchNormalization(),
        MaxPooling1D(pool_size=2),

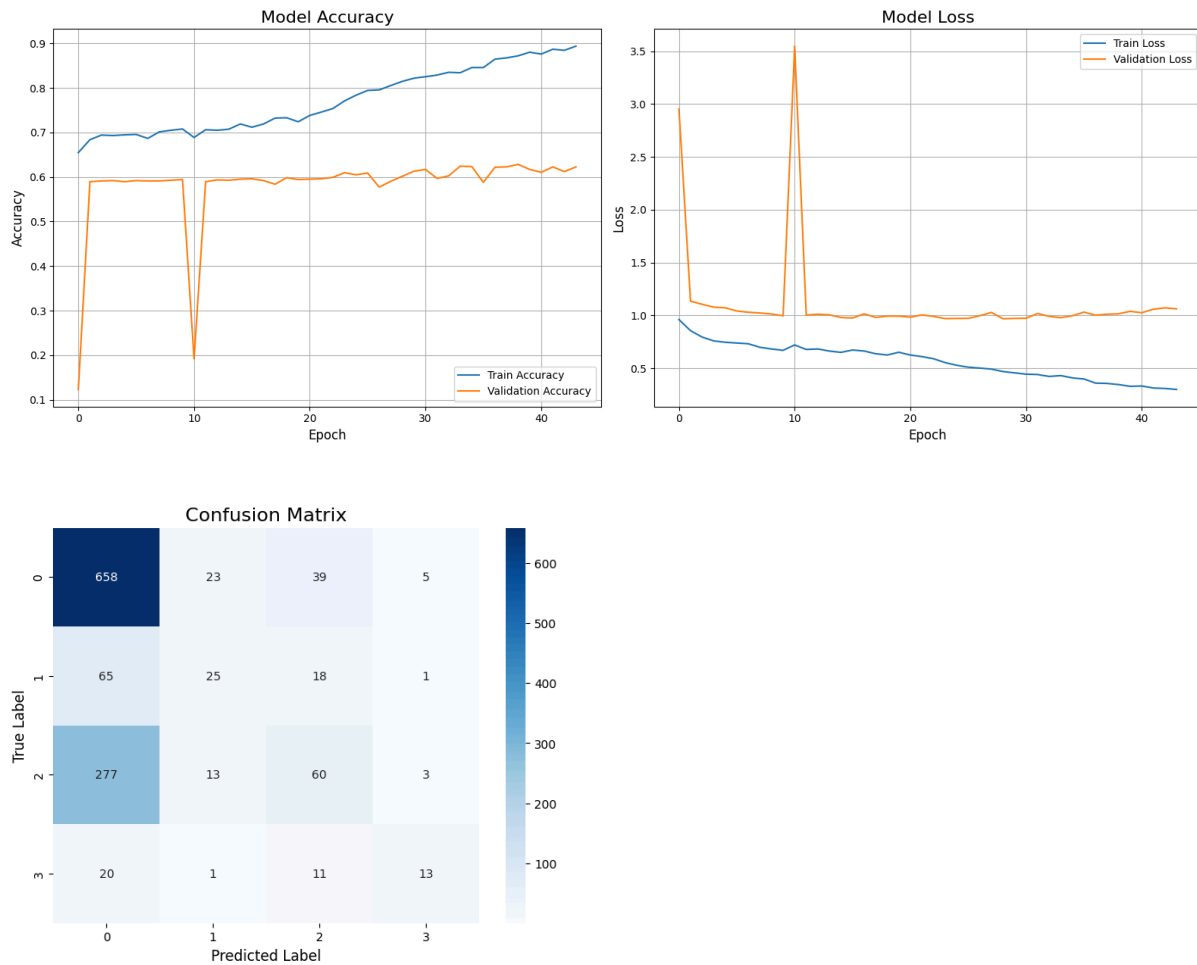
        Conv1D(64, kernel_size=3, activation='relu', padding='same',
              kernel_regularizer=l2(l2_reg)),
        BatchNormalization(),
        MaxPooling1D(pool_size=2),

        Conv1D(128, kernel_size=3, activation='relu', padding='same',
              kernel_regularizer=l2(l2_reg)),
        BatchNormalization(),
        MaxPooling1D(pool_size=2),
        # Timeline model (Bi-LSTM)
        Bidirectional(LSTM(64, return_sequences=False)),
        # Classification
        Dense(64, activation='relu', kernel_regularizer=l2(l2_reg)),
        Dropout(dropout_rate),
        Dense(n_classes, activation='softmax')
    ])
    return model
```

...

```
loss='categorical_crossentropy'
```

Results - does not work, just overfits to class 0:



Neural network #2 - I tried with focal loss:

```
def resnet_block(x, filters, kernel_size, l2_reg):
    y = Conv1D(filters, kernel_size, padding='same',
kernel_regularizer=l2(l2_reg))(x)
    y = BatchNormalization()(y)
    y = Activation('relu')(y)
    y = Conv1D(filters, kernel_size, padding='same',
kernel_regularizer=l2(l2_reg))(y)
    y = BatchNormalization()(y)
    if x.shape[-1] != filters:
        shortcut = Conv1D(filters, kernel_size=1, padding='same',
kernel_regularizer=l2(l2_reg))(x)
        shortcut = BatchNormalization()(shortcut)
    else:
        shortcut = x
    res = Add()([shortcut, y])
    res = Activation('relu')(res)
    return res
```

```
def build_resnet_lstm_model(input_shape, n_classes, l2_reg, dropout_rate):
    inputs = Input(shape=input_shape)

    # Feature extraction (ResNet)
    x = resnet_block(inputs, 64, 5, l2_reg)
    x = MaxPooling1D(pool_size=2)(x)
    x = resnet_block(x, 128, 3, l2_reg)
    x = MaxPooling1D(pool_size=2)(x)
    x = resnet_block(x, 256, 3, l2_reg)
    x = MaxPooling1D(pool_size=2)(x)

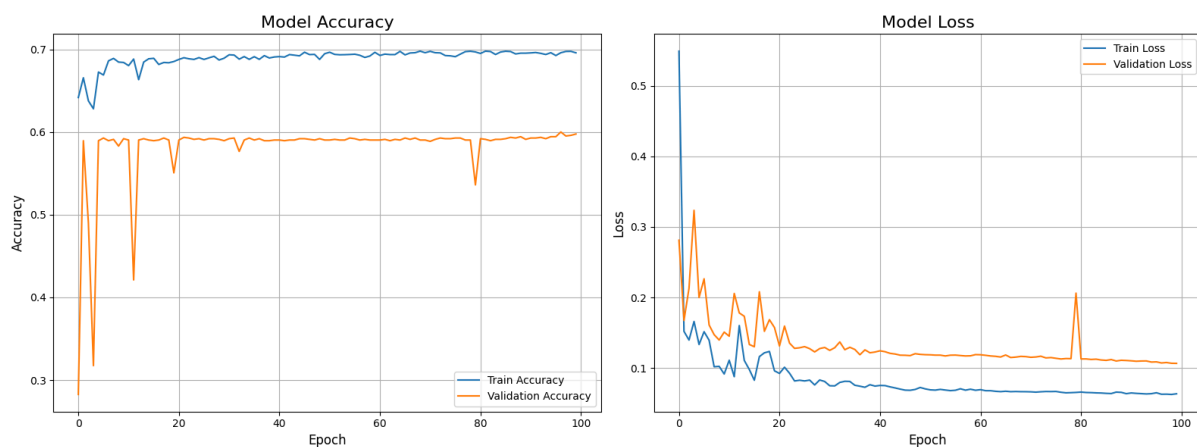
    # Timeline (Bi-LSTM)
    x = Bidirectional(LSTM(128, return_sequences=False))(x)

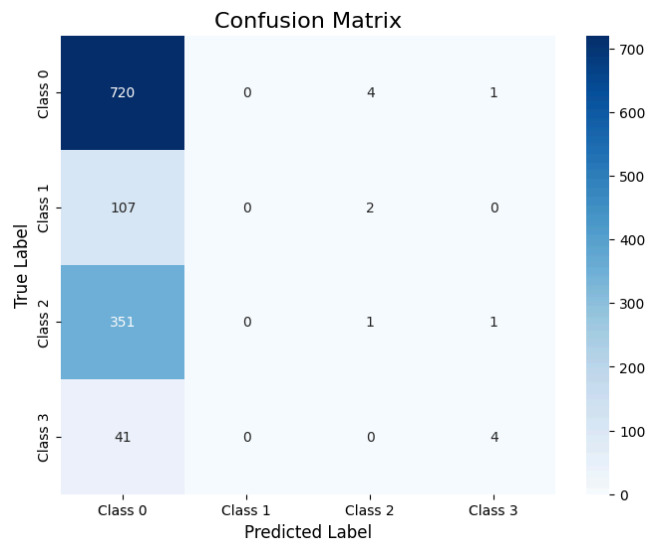
    # Classification
    x = Dense(128, activation='relu', kernel_regularizer=l2(l2_reg))(x)
    x = Dropout(dropout_rate)(x)
    outputs = Dense(n_classes, activation='softmax')(x)

    model = Model(inputs, outputs)
    return model

def focal_loss(gamma=2.0, alpha=0.25):
    def focal_loss_fixed(y_true, y_pred):
        y_true = tf.cast(y_true, tf.float32)
        epsilon = K.epsilon()
        y_pred = K.clip(y_pred, epsilon, 1. - epsilon)
        cross_entropy = -y_true * K.log(y_pred)
        loss = alpha * K.pow(1 - y_pred, gamma) * cross_entropy
        return K.sum(loss, axis=-1)
    return focal_loss_fixed
```

But it did not work at all - does not converge and overfits to class 0:





At this moment, I think I decided to use features and hopefully that worked well enough, with much less compute power (also getting T4 at 4 free hours/day was logistically difficult - it was available around 9 pm - midnight, so I opted for lighter methods).