## Contents

1 Getting started ................................................. 3
2 Citing our work ............................................... 5
3 Licence .......................................................... 7
4 Contributors .................................................... 9
5 Indices and tables ............................................. 37
Bibliography ....................................................... 39
Python Module Index ........................................... 41
Index .............................................................. 43
Extracting meaningful health information from large accelerometer datasets

A tool to extract meaningful health information from large accelerometer datasets. The software generates time-series and summary metrics useful for answering key questions such as how much time is spent in sleep, sedentary behaviour, or doing physical activity.

Dependancies include: unix, java 8 (Java 8 JDK) and python 3.7 (Anaconda’s Python 3 or installation via Brew should do the trick).

$ git clone git@github.com:activityMonitoring/biobankAccelerometerAnalysis.git
$ bash utilities/downloadDataModels.sh
$ pip3 install --user .
$ javac -cp java/JTransforms-3.1-with-dependencies.jar java/*.java
To extract a summary of movement (average sample vector magnitude) and (non)wear time from raw Axivity .CWA (or gzipped .cwa.gz) accelerometer files:

```bash
$ python3 accProcess.py data/sample.cwa.gz
<output written to data/sample-outputSummary.json>
<time series output written to data/sample-timeSeries.csv.gz>
```

The main output JSON will look like:

```json
{
    "file-name": "sample.cwa.gz",
    "acc-overall-avg(mg)": 32.78149,
    "wearTime-overall(days)": 5.8,
    "nonWearTime-overall(days)": 0.04,
    "quality-goodWearTime": 1
}
```

To visualise the time series and activity classification output:

```bash
python3 accPlot.py data/sample-timeSeries.csv.gz data/sample-plot.png
<output plot written to data/sample-plot.png>
```

The underlying modules can also be called in custom python scripts:

```python
from accelerometer import summariseEpoch
summary = {}
epochData, labels = summariseEpoch.getActivitySummary("sample-epoch.csv.gz", "sample-nonWear.csv.gz", summary)
# <nonWear file written to "sample-nonWear.csv.gz" and dict "summary" updated
# with outcomes>
```
Fig. 1: Output plot of overall activity and class predictions for each 30sec time window
When describing or using the UK Biobank accelerometer dataset, or using this tool to extract overall activity from your accelerometer data, please cite [Doherty2017].

When using this tool to extract sleep duration and physical activity behaviours from your accelerometer data, please cite [Willets2018] and [Doherty2018].
This project is released under a BSD 2-Clause Licence (see LICENCE file).
Contributors

Aiden Doherty, Sven Hollowell, Matthew Willetts (University of Oxford) Dan Jackson, Nils Hammerla (Newcastle University)

4.1 Usage

Our tool uses published methods to extract summary sleep and activity statistics from raw binary accelerometer data files.

4.1.1 Basic usage

To extract a summary of movement (average sample vector magnitude) and (non)wear time from raw Axivity .CWA accelerometer files:

```
$ python3 accProcess.py data/sample.cwa.gz
$ <summary output written to data/sample-outputSummary.json>
$ <time series output written to data/sample-timeSeries.csv.gz>
$ <non wear episodes output written to data/sample-nonWearEpisodes.csv.gz>
```

This may take a few minutes. When done, there will be four files (default in the same folder as the input .cwa file) containing the extracted data.
### File Information

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output-Summary.json</td>
<td>Summary statistics for the entire input file, such as data quality, acceleration and non-wear time grouped hour of day, and histograms of acceleration levels. Download a sample file.</td>
</tr>
<tr>
<td>Time-Series.csv</td>
<td>Acceleration magnitude for each epoch, and whether the data was imputed or not.</td>
</tr>
<tr>
<td>Epoch.csv</td>
<td>Acceleration data grouped in epochs (default = 5sec). Detailed information about XYZ acceleration, standard deviation, temperature, and data errors can be found in this file.</td>
</tr>
<tr>
<td>NonWear-Bouts.csv</td>
<td>Start and end times for any non-wear bouts, and the detected (presumably low) acceleration levels for each bout.</td>
</tr>
</tbody>
</table>

To visualise the time output:

```bash
$ python3 accPlot.py data/sample-timeSeries.csv.gz data/sample-plot.png
<output plot written to data/sample-plot.png>
```

![Output plot](data/sample-plot.png)

**Fig. 1:** Output plot of overall activity and class predictions for each 30sec time window

### 4.1.2 Input file types

**GENEActiv**

Process data from raw GENEActiv .bin files:
$ python3 accProcess.py data/sample.bin

**Actigraph**

Process data from raw Actigraph .gt3x files (both versions 1 and 2):

$ python3 accProcess.py data/sample.gt3x --sampleRate 80

An example Actigraph file can be obtained from the AGread gitHub page:

$ mv 119AGBPFLW\(2016-03-08\).gt3x data/actigraph-example.gt3x
$ python3 accProcess.py data/sample.gt3x --sampleRate 80

**CSV**

Process data from raw gzipped CSV files:

$ python3 accProcess.py data/sample.csv.gz

It is very unwise to store accelerometer data in .csv format. However, if one were to unzip and view .csv.gz file it would ideally be in this format:

$ wget "http://gas.ndph.ox.ac.uk/aidend/accModels/sample-small.csv.gz"
$ mv sample-small.csv.gz data/
$ gunzip data/sample.csv.gz
$ head -3 data/sample.csv

time,x,y,z
2014-05-07 13:29:50.439+0100 [Europe/London],-0.514,0.07,1.671
2014-05-07 13:29:50.449+0100 [Europe/London],-0.089,-0.805,-0.59

If your CSV is in a different format, there are options to flexibly parse these. Consider the below file with a different time format and the x/y/z columns having different index positions

$ head data/awkwardfile.csv
time,temperature, z,y,x
2014-05-07 13:29:50.439,20,0.07,1.671,-0.514
2014-05-07 13:29:50.449,20,-0.805,-0.59,-0.089

The above file can be processed as follows:

$ python3 accProcess.py data/awkwardFile.csv \
--csvTimeFormat 'yyyy-MM-dd HH:mm:ss.SSS' --csvTimeXYZColsIndex 0 4 2 3

### 4.1.3 Processing multiple files

To process multiple files, we recommend the following directory structure be used:

```
<studyName>/
  files.csv #listing all files in rawData directory (optional)
  rawData/ #all raw .cwa .cwa.gz .bin .gt3x files (no spaces in filename)
```

(continues on next page)
This can be created calling our utility script:

```
$ bash utilities/createStudyDir.sh /myStudy/
```

Next move relevant raw accelerometer files to the rawData folder:

```
$ mv *myAccelerometerFiles.cwa /myStudy/rawData/
```

Then use our python utility function to write processing cmds for all files:

```
from accelerometer import accUtils
accUtils.writeStudyAccProcessCmds("/myStudy/", "process-cmds.txt", 
    runName="dec18")
# <list of processing commands written to "process-cmds.txt">

# if for some reason we wanted to use different thresholds for moderate
# and vigorous intensity activities, we could go with
accUtils.writeStudyAccProcessCmds("/myStudy/", "process-cmds.txt", 
    runName="dec18", cmdOptions="--mgCutPointMVPA 90 --mgCutPointVPA 435")
# <list of processing commands written to "process-cmds.txt">
```

Note that if we don’t have `files.csv` in the existing directory, the utility function will automatically create a `files.csv` that contains the names of all the files in `rawData/`. For this to work, we need to specify which file type to use by setting the `accExt` parameter, e.g., cwa, CWA, bin, BIN, gt3x. We can also directly create our own `files.csv` with a column whose column name needs to be ‘fileName’.

We can then kick-start the processing of all accelerometer files. More advanced users will probably want to parallelise the below script using their HPC architecture of choice:

```
$ bash process-cmds.txt
```

Next, using our python utility function, we would like to collate all individual processed .json summary files into a single large csv for subsequent health analyses:

```
from accelerometer import accUtils
accUtils.collateJSONfilesToSingleCSV("/myStudy/summary/dec18/", 
    "myStudy/dec18-summary-info.csv")
# <summary CSV for all participants written to "/myStudy/dec18-summary-info.csv">
```

### Quality control

If is often necessary to check that all files have successfully processed. Our python utility function can write to file all participants’ data that was not successfully processed:

```
from accelerometer import accUtils
accUtils.identifyUnprocessedFiles("/myStudy/files.csv", "myStudy/dec18-summary-info.csv")
```

(continues on next page)
On other occasions some participants’ data may not have been calibrated properly. Our python utility function can assign the calibration coefs from a previous good use of a given device in the same study dataset:

```python
from accelerometer import accUtils
accUtils.updateCalibrationCoefs("myStudy/dec18-summary-info.csv", "myStudy/files-recalibration.csv")
```

Our python utility function can then re-write processing cmds as follows:

```python
from accelerometer import accUtils
accUtils.writeStudyAccProcessCmds("/myStudy/", "process-cmds-recalibration.txt", runName="dec18", filesID="files-calibration.csv", cmdOptions="--skipCalibration=True")
```

These ‘reprocessed’ files can then be processed as outlined in the section above.

### 4.1.4 Classifying different activity types

Note that a major fix/improvement was introduced in April 2020. You therefore need to download the updated files to achieve this.

```
$ git pull
$ bash utilities/downloadDataModels.sh
$ pip3 install --user .
$ javac -cp java/JTransforms-3.1-with-dependencies.jar java/*.java
```

Different activity classification models can be specified to identify different activity types. For example, to use activity states from the Willetts 2018 Scientific Reports paper:

```
```

To visualise the time series and new activity classification output:

```
$ python3 accPlot.py data/sample-timeSeries.csv.gz data/sample-plot.png --activityModel activityModels/willetts2018-apr20Update.tar
<output plot written to data/sample-plot.png>
```

**Training a bespoke model**

It is also possible to train a bespoke activity classification model. This requires a labelled dataset (.csv file) and a list of features (.txt file) to include from the epoch file.

First we need to evaluate how well the model works on unseen data. We therefore train a model on a ‘training set’ of participants, and then test how well that model works on a ‘test set’ of participant. The command below allows us to achieve this by specifying the test participant IDs (all other IDs will automatically go to the training set). This will output <participant, time, actual, predicted> predictions for each instance of data in the test set to a CSV file to help assess the model:

```
```
Fig. 2: Output plot of class predictions using Willetts 2018 classification model. Note different set of activity classes.
import accelerometer
accelerometer.accClassification.trainClassificationModel( 
    "activityModels/labelled-acc-epochs.csv", 
    featuresTxt="activityModels/features.txt", 
    testParticipants="4,5", 
    outputPredict="activityModels/test-predictions.csv", 
    rfTrees=1000, rfThreads=1)
# <Test predictions written to: activityModels/test-predictions.csv>

A number of metrics can then be calculated from the test predictions csv file:

import pandas as pd
from accelerometer import accClassification

# load data
d = pd.read_csv("test-predictions.csv")

# print summary to HTML file
htmlFile = "classificationReport.html"
yTrueCol = 'label'
yPredCol = 'predicted'
participantCol = 'participant'
accClassification.perParticipantSummaryHTML(d, yTrueCol, yPredCol, participantCol, htmlFile)

After evaluating the performance of our model on unseen data, we then re-train a final model that includes all possible data. We therefore specify the outputModel parameter, and also set testParticipants to ‘None’ so as to maximise the amount of training data for the final model. This results in an output .tar model:

import accelerometer
accelerometer.accClassification.trainClassificationModel( 
    "activityModels/labelled-acc-epochs.csv", 
    featuresTxt="activityModels/features.txt", 
    rfTrees=1000, rfThreads=1, 
    testParticipants=None, 
    outputModel="activityModels/sample-model.tar")
# <Model saved to activityModels/sample-model.tar>

This new model can be deployed as follows:

$ python3 accProcess.py --activityModel activityModels/sample-model.tar \
   data/sample.cwa.gz

Leave one out classification

To rigorously test a model with training data from <200 participants, leave one participant out evaluation can be helpful. Building on the above examples of training a bespoke model, we use python to create a list of commands to test the performance of a model trained on unseen data for each participant:

import pandas as pd
trainingFile = "activityModels/labelled-acc-epochs.csv"
d = pd.read_csv(trainingFile, usecols=['participant'])
pts = sorted(d['participant'].unique())

w = open('training-cmds.txt','w')
(continues on next page)
for p in pts:
    cmd = "import accelerometer;"
    cmd += "accelerometer.accClassification.trainClassificationModel(""
    cmd += "'" + trainingFile + ",""
    cmd += "featuresTxt='activityModels/features.txt',"
    cmd += "testParticipants='" + str(p) + ","
    cmd += "labelCol='label',"
    cmd += "outputPredict='activityModels/testPredict-' + str(p) + ",.csv',"
    cmd += "rfTrees=100, rfThreads=1"
    w.write('python3 -c "$"" + cmd + "$"" + \n')
w.close()

# <list of processing commands written to "training-cmds.txt">

These commands can be executed as follows:

$ bash training-cmds.txt

After processing the train/test commands, the resulting predictions for each test participant can be collated as follows:

$ head -1 activityModels/testPredict-1.csv > header.csv
$ awk 'FNR > 1' activityModels/testPredict-*.csv > tmp.csv
$ cat header.csv tmp.csv > test-predictions.csv
$ rm header.csv
$ rm tmp.csv

As indicated just above (under 'Training a bespoke model'), a number of metrics can be calculated for the 'testPredict-all.csv' file.

### 4.1.5 Advanced usage

To list all available processing options and their defaults, simply type:

$ python3 accProcess.py -h

Some example usages:

Specify file in another folder (note: use "" for file names with spaces):

$ python3 accProcess.py "/otherPath/other file.cwa"

Change epoch length to 60 seconds:

$ python3 accProcess.py data/sample.cwa.gz --epochPeriod 60

Manually set calibration coefficients:

$ python3 accProcess.py data/sample.cwa.gz --skipCalibration True \ 
   --calOffset -0.2 -0.4 1.5 --calSlope 0.7 0.8 0.7 \ 
   --calTemperature 0.2 0.2 0.2 --meanTemp 20.2

Extract calibrated and resampled raw data .csv.gz file from raw .cwa file:

$ python3 accProcess.py data/sample.cwa.gz --rawOutput True \ 
   --activityClassification False

The underlying modules can also be called in custom python scripts:
from accelerometer import summariseEpoch
summary = {}
epochData, labels = summariseEpoch.getActivitySummary( \
    "data/sample-epoch.csv.gz", "data/sample-nonWear.csv.gz", summary)
# <nonWear file written to "data/sample-nonWear.csv.gz" and dict "summary" \n# updated with outcomes>

4.2 Methods

Interpreted levels of physical activity can vary, as many approaches can be taken to extract summary physical activity information from raw accelerometer data. To minimise error and bias, our tool uses published methods to calibrate, resample, and summarise the accelerometer data e.g. [Doherty2017] [Willetts2018] and [Doherty2018].

Fig. 3: UK Biobank triaxial accelerometer and processing steps to extract physical activity information. Axivity AX3 triaxial accelerometer worn on dominant hand as used in UK Biobank (top left). Time series trace of processed accelerometer values after one week of wear (top right). Overview of process to extract proxy physical activity information from raw accelerometer data (bottom).

4.2.1 Data preparation

Calibration

To ensure different devices provide a similar output under similar conditions we calibrate the acceleration signals to local gravity using a procedure initially described by [vanHees2014] and run in UK Biobank [Doherty2017].

Briefly, we identify stationary periods in ten second windows where all three axes have a standard deviation of less than 13.0 mg. These stationary periods are then used to optimise the gain and offset for each axis (6 parameters) to fit a unit gravity sphere using ordinary least squares linear regression.

Highlight interrupts, and invalid values

Clipped values, which occur when the sensor’s dynamic range of ±8g is exceeded, are flagged before and after calibration. Recording errors and ‘interrupts’, which could occur for example if a participant tried to plug their accelerometer device into a computer, are also logged.
Resampling

While the accelerometer is setup to record data at 100Hz, the actual sample rate can fluctuate between 94-104Hz. The implication of this is the introduction of differential effects between individuals when extracting frequency domain features for activity classification. Thus, valid data is resampled to 100 Hz using linear interpolation, except for interrupts lasting longer than 1 second which are set to missing.

4.2.2 Vector magnitude processing

Combine x/y/z axes

We compute the sample level Euclidean norm of the acceleration in x/y/z axes.

Gravity and noise removal

Machine noise is removed using a fourth order Butterworth low pass filter with a cutoff frequency of 20Hz. This filter is applied to the vector magnitude scores, rather than the individual axes, due to more precisely capturing arm rotations. In order to separate out the activity-related component of the acceleration signal, we remove one gravitational unit from the vector magnitude, with remaining negative values truncated to zero. We used this approach in UK Biobank [Doherty2017] and it has been validated against doubly labelled water, see [vanHees2011] and [White2018].

Epoch generation

To describe the overall level and distribution of physical activity intensity, we combined the sample level data into five second epochs for summary data analysis, maintaining the average vector magnitude value over the epoch. To represent the distribution of time spent by an individual in different levels of physical activity intensity, we generate an empirical cumulative distribution function from all available epochs.

4.2.3 Activity classification

Feature extraction

For every non-overlapping 30-second time window (default epoch period our model is trained on), we extracted a 126-dimensional feature vector. These time and frequency domain features includ: vector magnitude, it’s mean, standard deviation, coefficient of variation, median, min, max, 25th & 75th percentiles, mean amplitude deviation, mean power deviation, kurtosis & skew, and Fast Fourier Transform (FFT) 1–15Hz. Features also includ the following in each individual axis of movement: mean, range, standard deviation, covariance, and FFT 1–15Hz. Roll, pitch, yaw, x/y/z correlations, frequency and power bands are also extracted [Willetts2018] [Doherty2018]

Classification

For activity classification we use a two stage model consisting of balanced random forests and hidden markov models.

Balanced random forests

Balanced random forests offer a powerful nonparametric discriminative method for multi-activity classification. Predictions of a random forest are an aggregate of individual CART trees (Classification And Regression Trees). CART trees are binary trees consisting of split nodes and terminal leaf nodes. In our case, each tree is constructed from a
training set of feature data (just described above) along with ground truth activity classes (free living camera data in [Willetts2018] [Doherty2018]).

There is randomness in the model, as we only give each tree a subset of data and features. This ensures that the trees have low correlation and is necessary as the CART algorithm itself is deterministic. Given the unbalanced nature of our dataset, where some behaviours occur rarely, we use balanced Random Forests to train each tree with a balanced subset of training data. If we have n_rare instances of the rarest class, we pick n_rare samples, with replacement, of data of each of our classes to form our training set for each tree. As each tree is given only a small fraction of data, we make many more trees than in a standard random forest so that the same number of data points are sampled in training as with a standard application of random forests [Willetts2018].

**Hidden Markov models**

Random forests are able to classify datapoints, but do not have an understanding of our data as having come from a time series. Therefore we use a hidden Markov model (HMM) to encode the temporal structure of the sequence of classes and thus obtain a more accurate sequence of predicted classes. The transition matrix (likelihood of moving from one activity type to another) and emission distribution (likelihood of random forest correctly classifying a given activity type) are empirically calculated. The transition matrix is calculated from the training set sequence of activity states. The calculation of emission probabilities comes from the out of bag class votes of the random forest. Recall that in a random forest each tree is trained on a subset of the training data. Thus by passing through each tree the training data that it was not trained on we get an estimate of the error of the forest. This gives us directly the probability of predicting each class given the true activity class [Willetts2018].

With this empirically defined HMM, we can then run the Viterbi algorithm to find the most likely sequence of states given a sequence of observed emissions from the random forest. This smoothing corrects erroneous predictions from the random forest, such as where the error is a blip of one activity surrounded by another and the transitions between those two classes of activity are rare.

![Fig. 4: Diagram of a Hidden Markov Model.](image)

**4.2.4 Physical activity analysis**

**Detect non-wear**

We remove non-wear time, defined as consecutive stationary episodes lasting for at least 60 minutes. The same standard deviation threshold criteria are applied as described in the calibration procedure to identify stationary episodes from the selected epochs.
Wear-time weighting

We impute non-wear data segments using the average of similar time-of-day vector magnitude and intensity distribution data points with one minute granularity on different days of the measurement. This imputation accounts for potential wear time diurnal bias where, for example, if the device was systematically not worn during sleep in an individual, the crude average vector magnitude during wear time would be a biased overestimate of the true average.

![Graph showing imputation for non-wear data](image)

Fig. 5: Example imputation for non-wear (blue shaded) data.

### 4.2.5 Summary physical activity variable

#### Minimum wear time

A physical activity outcome variable is generated by averaging all worn and imputed values. For analysis of UK Biobank accelerometer data, it may be prudent to remove individuals who had less than three days (72 hours) of data or who did not have data in each one-hour period of the 24-hour cycle. We defined these minimum wear time guidelines by performing missing data simulations on 29,765 participants [Doherty2017]. Using intraclass correlation coefficients, at least 72 hours (3 days) of wear were needed to be within 10% of the true stable seven day measure.

#### Time series file

A .csv time series file is generated for each participant. This provides researchers with a simple way to interrogate the epoch level data for each physical activity outcome variable, without the need for expertise in processing large complex raw data files.

### 4.2.6 References

### 4.3 accelerometer package

#### 4.3.1 Submodules

#### 4.3.2 accelerometer.accClassification module

Module to support machine learning of activity states from acc data
Perform classification of activity states from epoch feature data

Based on a balanced random forest with a Hidden Markov Model containing transitions between predicted activity states and emissions trained using a free-living groundtruth to identify pre-defined classes of behaviour from accelerometer data.

**Parameters**

- `epochFile` *(str)* – Input csv file of processed epoch data
- `activityModel` *(str)* – Input tar model file which contains random forest pickle model, HMM priors/transitions/emissions npy files, and npy file of METs for each activity state

**Returns**

- Pandas dataframe of activity epoch data with one-hot encoded labels
- Activity state labels

**accelerometer.accClassification.addReferenceLabelsToNewFeatures** *(featuresFile, referenceLabelsFile, outputFile, featuresTxt='activityModels/features.txt', labelCol='label', participantCol='participant', atomicLabelCol='annotation', metCol='MET')*

Append reference annotations to newly extracted feature data

This method helps add existing curated labels (from referenceLabelsFile) to a file with newly extracted features (both pre-sorted by participant and time).

**Parameters**

- `featuresFile` *(str)* – Input csv file of new features data, pre-sorted by time
- `referenceLabelsFile` *(str)* – Input csv file of reference labelled data, pre-sorted by time
- `outputFile` *(str)* – Output csv file of new features data with reference labels
- `featuresTxt` *(str)* – Input txt file listing feature column names
- `labelCol` *(str)* – Input label column
- `participantCol` *(str)* – Input participant column
- `atomicLabelCol` *(str)* – Input ‘atomic’ annotation e.g. ‘walking with dog’ vs. ‘walking’
- `metCol` *(str)* – Input MET column

**Returns**

- New csv file written to `<outputFile>`

**Return type** void
Example

```python
>>> from accelerometer import accClassification
>>> accClassification.addReferenceLabelsToNewFeatures("newFeats.csv", "refLabels.csv", "newFeatsPlusLabels.csv")
<file written to newFeatsPlusLabels.csv>
```

`accelerometer.accClassification.getFileFromTar(tarArchive, targetFile)`
Read file from tar

This is currently more tricky than it should be see https://github.com/numpy/numpy/issues/7989

**Parameters**

- `tarArchive (str)` – Input tarfile object
- `targetFile (str)` – Target individual file within .tar

**Returns** file object byte stream

**Return type** object

`accelerometer.accClassification.getListFromTxtFile(inputFile)`
Read list of items from txt file and return as list

**Parameters** `inputFile (str)` – Input file listing items

**Returns** list of items

**Return type** list

`accelerometer.accClassification.perParticipantSummaryHTML(dfParam, yTrueCol, yPredCol, pidCol, outHTML)`
Provide HTML summary of how well activity classification model works at the per-participant level

**Parameters**

- `dfParam (dataframe)` – Input pandas dataframe
- `yTrueCol (str)` – Input for y_true column label
- `yPreadCol (str)` – Input for y_pred column label
- `pidCol (str)` – Input for participant ID column label
- `outHTML (str)` – Output file to print HTML summary to

**Returns** HTML file reporting kappa, accuracy, and confusion matrix

**Return type** void

`accelerometer.accClassification.saveModelsToTar(tarArchive, featureCols, rfModel, priors, transitions, emissions, METs, featuresTxt='featureCols.txt', rfModelFile='rfModel.pkl', hmmPriors='hmmPriors.npy', hmmEmissions='hmmEmissions.npy', hmmTransitions='hmmTransitions.npy', hmmMETs='METs.npy')`

Save random forest and hidden markov models to tarArchive file

Note we must use the same version of python and scikit learn as in the intended deployment environment

**Parameters**
Accelerometer Documentation, Release 2.0

- **tarArchive (str)** – Output tarfile
- **featureCols (list)** – Input list of feature columns
- **rfModel (sklearn.RandomForestClassifier)** – Input random forest model
- **priors (numpy.array)** – Input prior probabilities for each activity state
- **transitions (numpy.array)** – Input probability matrix of transitioning from one activity state to another
- **emissions (numpy.array)** – Input probability matrix of RF prediction being true
- **METs (numpy.array)** – Input array of average METs per activity state
- **featuresTxt (str)** – Intermediate output txt file of features
- **rfModelFile (str)** – Intermediate output random forest pickle model
- **hmmPriors (str)** – Intermediate output HMM priors npy
- **hmmEmissions (str)** – Intermediate output HMM emissions npy
- **hmmTransitions (str)** – Intermediate output HMM transitions npy
- **hmmMETs (str)** – Intermediate output HMM METs npy

**Returns**

tar file of RF + HMM written to tarArchive

**Return type**

void

```python
accelerometer.accClassification.trainClassificationModel(
    trainingFile, labelCol='label', participantCol='participant',
    atomicLabelCol='annotation', metCol='MET', featuresTxt='activityModels/features.txt',
    trainParticipants=None, testParticipants=None, rfThreads=1,
    rfTrees=1000, rfFeats=None, rfDepth=None,
    outputPredict='activityModels/test-predictions.csv', outputModel=None)
```

Train model to classify activity states from epoch feature data

Based on a balanced random forest with a Hidden Markov Model containing transitions between predicted activity states and emissions trained using the input training file to identify pre-defined classes of behaviour from accelerometer data.

**Parameters**

- **trainingFile (str)** – Input csv file of training data, pre-sorted by time
- **labelCol (str)** – Input label column
- **participantCol (str)** – Input participant column
- **atomicLabelCol (str)** – Input ‘atomic’ annotation e.g. ‘walking with dog’ vs. ‘walking’
- **metCol (str)** – Input MET column
• **featuresTxt** *(str)* – Input txt file listing feature column names
• **trainParticipants** *(str)* – Input comma separated list of participant IDs to train on.
• **testParticipants** *(str)* – Input comma separated list of participant IDs to test on.
• **rfThreads** *(int)* – Input num threads to use when training random forest
• **rfTrees** *(int)* – Input num decision trees to include in random forest
• **outputPredict** *(str)* – Output CSV of person, label, predicted
• **outputModel** *(str)* – Output tarfile object which contains random forest pickle model, HMM priors/transitions/emissions npy files, and npy file of METs for each activity state. Will only output trained model if this is not null e.g. “activityModels/sample-model.tar”

**Returns** New model written to <outputModel> OR csv of test predictions written to <outputPredict>

**Return type**  void

```
accelerometer.accClassification.train_HMM(rfModel, y_trainF, labelCol)
```

Train Hidden Markov Model

Use data not considered in construction of random forest to estimate probabilities of: i) starting in a given state; ii) transitioning from one state to another; and iii) probability of the random forest being correct when predicting a given class (emission probability)

**Parameters**

• **rfModel** *(sklearn.RandomForestClassifier)* – Input random forest object
• **y_trainF** *(dataframe.Column)* – Input groundtruth for each instance
• **labelCol** *(str)* – Input label column

**Returns**

- **states** - List of unique activity state labels
  rtype: numpy.array
- **priors** - Prior probabilities for each activity state
  rtype: numpy.array
- **transitions** - Probability matrix of transitioning from one activity state to another
  rtype: numpy.array
- **emissions** - Probability matrix of RF prediction being true
  rtype: numpy.array

```
accelerometer.accClassification.viterbi(observations, states, priors, transitions, emissions, probabilistic=False)
```

Perform HMM smoothing over observations via Viterbi algorithm

**Parameters**

• **observations** *(list(str))* – List/sequence of activity states
• **states** *(numpy.array)* – List of unique activity state labels
• **priors** *(numpy.array)* – Prior probabilities for each activity state
• **transitions** *(numpy.array)* – Probability matrix of transitioning from one activity state to another
• **emissions** *(numpy.array)* – Probability matrix of RF prediction being true
• **probabilistic** *(bool)* – Write probabilistic output for each state, rather than writing most likely state for any given prediction.

**Returns**  Smoothed list/sequence of activity states

**Return type**  list(str)

**accelerometer.accClassification.wristListToTxtFile**(inputList, outputFile)

Write list of items to txt file

**Parameters**

- **inputList** *(list)* – input list
- **outputFile** *(str)* – Output txt file

**Returns**  list of feature columns

**Return type**  void

### 4.3.3 accelerometer.accUtils module

Module to provide generic utilities for other accelerometer modules.

**accelerometer.accUtils.collateJSONfilesToSingleCSV**(inputJsonDir, outputCsvFile)

read all summary *.json files and convert into one large CSV file

Each json file represents summary data for one participant. Therefore output CSV file contains summary for all participants.

**Parameters**

- **inputJsonDir** *(str)* – Directory containing JSON files
- **outputCsvFile** *(str)* – Output CSV filename

**Returns**  New file written to <outputCsvFile>

**Return type**  void

**Example**

```python
>>> import accUtils

>>> accUtils.collateJSONfilesToSingleCSV("data/", "data/summary-all-files.csv")
<summary CSV of all participants/files written to "data/sumamry-all-files.csv">
```

**accelerometer.accUtils.createDirIfNotExists**(folder)

Create directory if it doesn’t currently exist

**Parameters**  folder *(str)* – Directory to be checked/created

**Returns**  Dir now exists (created if didn’t exist before, otherwise untouched)

**Return type**  void

**Example**

```python
>>> import accUtils

>>> accUtils.createDirIfNotExists("/myStudy/summary/dec18/")
<folder "/myStudy/summary/dec18/" now exists>
```

**accelerometer.accUtils.date_parser**(t)

Parse date a date string of the form e.g. 2020-06-14 19:01:15.123+0100 [Europe/London]
accelerometer.accUtils.date_strftime(t)
    Convert to time format of the form e.g. 2020-06-14 19:01:15.123+0100 [Europe/London]

accelerometer.accUtils.formatNum(num, decimalPlaces)
    return str of number formatted to number of decimal places

    When writing out 10,000’s of files, it is useful to format the output to n decimal places as a space saving measure.

    Parameters
    • num (float) – Float number to be formatted.
    • decimalPlaces (int) – Number of decimal places for output format

    Returns  Number formatted to number of decimalPlaces

    Return type  str

    Example

    >>> import accUtils
    >>> accUtils.formatNum(2.567, 2)
    2.57

accelerometer.accUtils.identifyUnprocessedFiles(filesCsv, summaryCsv, outputFilesCsv)
    identify files that have not been processed

    Look through all processed accelerometer files, and find participants who do not have records in the summary csv file. This indicates there was a problem in processing their data. Therefore, output will be a new .csv file to support reprocessing of these files.

    Parameters
    • filesCsv (str) – CSV listing acc files in study directory
    • summaryCsv (str) – Summary CSV of processed dataset
    • outputFilesCsv (str) – Output csv listing files to be reprocessed

    Returns  New file written to <outputCsvFile>

    Return type  void

    Example

    >>> import accUtils
    >>> accUtils.identifyUnprocessedFiles("study/files.csv", study/summary-all-files.csv", "study/files-reprocess.csv")
    <Output csv listing files to be reprocessed written to "study/files-reprocess.csv">

accelerometer.accUtils.meanCIstr(mean, std, n, numDecimalPlaces)
    return str of mean and 95% confidence interval numbers formatted

    Parameters
    • mean (float) – Mean number to be formatted.
    • std (float) – Standard deviation number to be formatted.
    • n (int) – Number of observations
    • decimalPlaces (int) – Number of decimal places for output format
accelerometer Documentation, Release 2.0

Returns String formatted to number of decimalPlaces
Return type str
Example

```python
>>> import accUtils
>>> accUtils.meanSDstr(2.567, 0.089, 2)
2.57 (0.09)
```

accelerometer.accUtils.meanSDstr(mean, std, numDecimalPlaces)
return str of mean and stdev numbers formatted to number of decimalPlaces

Parameters

- **mean** (float) – Mean number to be formatted.
- **std** (float) – Standard deviation number to be formatted.
- **decimalPlaces** (int) – Number of decimal places for output format

Returns String formatted to number of decimalPlaces
Return type str
Example

```python
>>> import accUtils
>>> accUtils.meanSDstr(2.567, 0.089, 2)
2.57 (0.09)
```

accelerometer.accUtils.toScreen(msg)
Print msg str prepended with current time

Parameters **msgs** (str) – Message to be printed to screen

Returns Print msg str prepended with current time
Return type void
Example

```python
>>> import accUtils
>>> accUtils.toScreen("hello")
2018-11-28 10:53:18 hello
```

accelerometer.accUtils.updateCalibrationCoefs(inputCsvFile, outputCsvFile)
read summary .csv file and update coefs for those with poor calibration

Look through all processed accelerometer files, and find participants that did not have good calibration data. Then assigns the calibration coeffs from previous good use of a given device. Output will be a new .csv file to support reprocessing of uncalibrated files with new pre-specified calibration coeffs.

Parameters

- **inputCsvFile** (str) – Summary CSV of processed dataset
- **outputCsvFile** (str) – Output CSV of files to be reprocessed with new calibration info

Returns New file written to <outputCsvFile>
Return type void
Example
```python
>>> import accUtils

accUtils.updateCalibrationCoefs("data/summary-all-files.csv", "study/files-recalibration.csv")

<CSV of files to be reprocessed written to "study/files-recalibration.csv">
```

```
accelerometer.accUtils.writeFilesWithCalibrationCoefs(inputCsvFile, outputCsvFile)

read summary .csv file and write files.csv with calibration coefs

Look through all processed accelerometer files, and write a new .csv file to support reprocessing of files with pre-specified calibration coefs.

Parameters

- **inputCsvFile** (*str*) – Summary CSV of processed dataset
- **outputCsvFile** (*str*) – Output CSV of files to process with calibration info

Returns

New file written to <outputCsvFile>

Return type

`void`

Example

```python
>>> import accUtils

>>> accUtils.writeFilesWithCalibrationCoefs("data/summary-all-files.csv", "study/files-calibrated.csv")

<CSV of files to be reprocessed written to "study/files-calibrated.csv">
```

```python
accelerometer.accUtils.writeStudyAccProcessCmds(studyDir, cmdFile, runName='default', accExt='cwa', cmdOptions=None, files='files.csv')

Read files to process and write out list of processing commands

This method assumes that a study directory structure has been created by the createStudyDir.sh script where there is a folder structure of <studyName>/

- files.csv #listing all files in rawData directory rawData/ #all .cwa .bin .gt3x files summary/ #to store outputSummary.json epoch/ #to store feature output for 30sec windows timeSeries/ #simple csv time series output (VMag, activity binary predictions) nonWear/ #bouts of nonwear episodes stationary/ #temp store for features of stationary data for calibration clusterLogs/ #to store terminal output for each processed file

If files.csv exists, process files listed here. If not, all files in rawData/ are read and listed in files.csv

Then an acc processing command is written for each file and written to cmdFile

Parameters

- **studyDir** (*str*) – Root directory of study
- **cmdFile** (*str*) – Output .txt file listing acc processing commands
- **runName** (*str*) – Name to assign to this processing run. Supports processing dataset in multiple different ways.
- **accExt** (*str*) – Acc file type e.g. cwa, CWA,bin, BIN, gt3x...
- **cmdOptions** (*str*) – String of processing options e.g. “–epochPeriod 10” Type ‘python3 accProcess.py -h’ for full list of options
- **files** (*str*) – Name of .csv file listing acc files to process

Returns

New file written to <cmdFile>
```
Return type void

Example

```python
>>> import accUtils
>>> accUtils.writeStudyAccProcessingCmds("/myStudy/", "myStudy-cmds.txt")
<cmd options written to "myStudy-cmds.txt”>
```

```python
accelerometer.accUtils.writeTimeSeries(e, labels, tsFile)
```


Parameters

- **labels** *(list(str)) – Activity state labels*
- **tsFile** *(dict) – output CSV filename*

Returns None

Return type void

### 4.3.4 accelerometer.circadianRhythms module

Module to support calculation of metrics of circadian rhythm from acc data

```python
accelerometer.circadianRhythms.calculateFourierFreq(e, epochPeriod, fourierWithAcc, labels, summary)
```

Calculate the most prevalent frequency in a fourier analysis

Parameters

- **e** *(pandas.DataFrame) – Pandas dataframe of epoch data*
- **epochPeriod** *(int) – Size of epoch time window (in seconds)*
- **labels** *(list(str)) – Activity state labels*
- **summary** *(dict) – Output dictionary containing all summary metrics*

Returns Write dict <summary> keys ‘fourier frequency-<1/days>’

```python
accelerometer.circadianRhythms.calculateM10L5(e, epochPeriod, summary)
```

Calculates the M10 L5 relative amplitude from the average acceleration from the ten most active hours and 5 least most active hours

Parameters

- **e** *(pandas.DataFrame) – Pandas dataframe of epoch data*
- **epochPeriod** *(int) – Size of epoch time window (in seconds)*
- **summary** *(dict) – Output dictionary containing all summary metrics*

Returns Write dict <summary> keys ‘M10 L5-<rel amp>’

```python
accelerometer.circadianRhythms.calculatePSD(e, epochPeriod, fourierWithAcc, labels, summary)
```

Calculate the power spectral density from fourier analysis of a 1 day frequency

Parameters
• \texttt{e (pandas.DataFrame)} – Pandas dataframe of epoch data
• \texttt{epochPeriod \text{(int)}} – Size of epoch time window (in seconds)

:param bool fourierWithAcc:True calculates fourier done with acceleration data instead of sleep data :param list(str) labels: Activity state labels :param dict summary: Output dictionary containing all summary metrics

Returns Write dict `<summary>` keys `PSD-<W/Hz>`

### 4.3.5 accelerometer.device module

Module to process raw accelerometer files into epoch data.

\texttt{accelerometer.device.getAxivityDeviceId (cwaFile)}

Get serial number of Axivity device

Parameters \texttt{cwaFile (str)} – Input raw .cwa accelerometer file

Returns Device ID

Return type int

\texttt{accelerometer.device.getCalibrationCoefs (staticBoutsFile, summary)}

Identify calibration coefficients from java processed file

Get axes offset/gain/temp calibration coefficients through linear regression of stationary episodes

Parameters

• \texttt{stationaryFile (str)} – Output/temporary file for calibration
• \texttt{summary (dict)} – Output dictionary containing all summary metrics

Returns Calibration summary values written to dict `<summary>`

Return type void

\texttt{accelerometer.device.getDeviceId (inputFile)}

Get serial number of device

First decides which DeviceId parsing method to use for `<inputFile>`.

Parameters \texttt{inputFile (str)} – Input raw accelerometer file

Returns Device ID

Return type int

\texttt{accelerometer.device.getGT3XDeviceId (gt3xFile)}

Get serial number of Actigraph device

 Parses the unique serial code from the header of a GT3X accelerometer file

Parameters \texttt{gt3xFile (str)} – Input raw .gt3x accelerometer file

Returns Device ID

Return type int

\texttt{accelerometer.device.getGeneaDeviceId (binFile)}

Get serial number of GENEActiv device

 Parses the unique serial code from the header of a GENEActiv accelerometer file

Parameters \texttt{binFile (str)} – Input raw .bin accelerometer file

Chapter 4. Contributors
Returns  Device ID
Return type  int

accelerometer.device.getOmconvertInfo (omconvertInfoFile, summary)

Identify calibration coefficients for omconvert processed file

Get axes offset/gain/temp calibration coeffs from omconvert info file

Parameters

- omconvertInfoFile (str) – Output information file from omconvert
- summary (dict) – Output dictionary containing all summary metrics

Returns  Calibration summary values written to dict <summary>
Return type  void

accelerometer.device.processInputFileToEpoch (inputFile, timeZone, timeShift, epochFile, stationaryFile, summary, skipCalibration=False, stationaryStd=13, xyzIntercept=[0.0, 0.0, 0.0], xyzSlope=[1.0, 1.0, 1.0], xyzTemp=[0.0, 0.0, 0.0], meanTemp=20.0, rawDataParser='AccelerometerParser', javaHeapSpace=None, useFilter=True, sampleRate=100, epochPeriod=30, activityClassification=True, rawOutput=False, rawFile=None, npyOutput=False, npyFile=None, startTime=None, endTime=None, verbose=False, csvStartTime=None, csvSampleRate=None, csvTimeFormat="yyyy-MM-dd HH:mm:ss.SSSxxxx ["VV"]", csvStartRow=1, csvTimeXYZColsIndex=None)

Process raw accelerometer file, writing summary epoch stats to file

This is usually achieved by

1) identify 10sec stationary epochs
2) record calibrated axes scale/offset/temp vals + static point stats
3) use calibration coefficients and then write filtered avgVm epochs to <epochFile> from <inputFile>

Parameters

- inputFile (str) – Input <cwa/cwa.gz/bin/gt3x> raw accelerometer file
- epochFile (str) – Output csv.gz file of processed epoch data
- stationaryFile (str) – Output/temporary file for calibration
- summary (dict) – Output dictionary containing all summary metrics
- skipCalibration (bool) – Perform software calibration (process data twice)
- stationaryStd (int) – Gravity threshold (in mg units) for stationary vs not
- xyzIntercept (list (float)) – Calibration offset [x, y, z]
- xyzSlope (list (float)) – Calibration slope [x, y, z]
• **xyzTemp** *(list (float)) – Calibration temperature coefficient [x, y, z]*

• **meanTemp** *(float) – Calibration mean temperature in file*

• **rawDataParser** *(str) – External helper process to read raw acc file. If a java class, it must omit .class ending.*

• **javaHeapSpace** *(str) – Amount of heap space allocated to java subprocesses. Useful for limiting RAM usage.*

• **useFilter** *(bool) – Filter ENMOtrunc signal*

• **sampleRate** *(int) – Resample data to n Hz*

• **epochPeriod** *(int) – Size of epoch time window (in seconds)*

• **activityClassification** *(bool) – Extract features for machine learning*

• **rawOutput** *(bool) – Output calibrated and resampled raw data to a .csv.gz file? requires ~50MB/day.*

• **rawFile** *(str) – Output raw data “.csv.gz” filename*

• **npyOutput** *(bool) – Output calibrated and resampled raw data to a .npy file? requires ~60MB/day.*

• **npyFile** *(str) – Output raw data “.npy” filename*

• **startTime** *(datetime) – Remove data before this time in analysis*

• **endTime** *(datetime) – Remove data after this time in analysis*

• **verbose** *(bool) – Print verbose output*

• **csvStartTime** *(datetime) – start time for csv file when time column is not available*

• **csvSampleRate** *(float) – sample rate for csv file when time column is not available*

• **csvTimeFormat** *(str) – time format for csv file when time column is available*

• **csvStartRow** *(int) – start row for accelerometer data in csv file*

• **csvTimeXYZColsIndex** *(str) – index of column positions for XYZT columns, e.g. “1,2,3,0”

**Returns** Raw processing summary values written to dict <summary>

**Return type** void

**Example**

```python
>>> import device
>>> summary = {}
>>> device.processInputFileToEpoch('inputFile.cwa', 'epochFile.csv.gz',
    'stationary.csv.gz', summary)
<epoch file written to "epochFile.csv.gz", and calibration points to
'stationary.csv.gz'>
```

**accelerometer.device.storeCalibrationInformation** *(summary, bestIntercept, bestSlope, bestTemp, meanTemp, initError, bestError, xMin, xMax, yMin, yMax, zMin, zMax, nStatic, calibrationSphereCriteria=0.3)*

Store calibration information to output summary dictionary

**Parameters**
**accelerometer Documentation, Release 2.0**

- **summary (dict)** – Output dictionary containing all summary metrics
- **bestIntercept (list (float))** – Best x/y/z intercept values
- **bestSlope (list (float))** – Best x/y/z slope values
- **bestTemperature (list (float))** – Best x/y/z temperature values
- **meanTemp (float)** – Calibration mean temperature in file
- **initError (float)** – Root mean square error (in mg) before calibration
- **initError** – Root mean square error (in mg) after calibration
- **xMin (float)** – xMin information on spread of stationary points
- **xMax (float)** – xMax information on spread of stationary points
- **yMin (float)** – yMin information on spread of stationary points
- **yMax (float)** – yMax information on spread of stationary points
- **zMin (float)** – zMin information on spread of stationary points
- **zMax (float)** – zMax information on spread of stationary points
- **nStatic (int)** – number of stationary points used for calibration
- **calibrationSphereCriteria (float)** – Threshold to check how well file was calibrated

**Returns**  Calibration summary values written to dict <summary>

**Return type**  void

accelerometer.device.storeCalibrationParams (summary, xyzOff, xyzSlope, xyzTemp, meanTemp)

Store calibration parameters to output summary dictionary

**Parameters**

- **summary (dict)** – Output dictionary containing all summary metrics
- **xyzOff (list (float))** – intercept [x, y, z]
- **xyzSlope (list (float))** – slope [x, y, z]
- **xyzTemp (list (float))** – temperature [x, y, z]
- **meanTemp (float)** – Calibration mean temperature in file

**Returns**  Calibration summary values written to dict <summary>

**Return type**  void

### 4.3.6 accelerometer.summariseEpoch module

Module to generate overall activity summary from epoch data.

accelerometer.summariseEpoch.calculateECDF (e, inputCol, summary, useRecommendedImputation)

Calculate activity intensity empirical cumulative distribution

The input data must not be imputed, as ECDF requires different imputation where nan/non-wear data segments are IMPUTED FOR EACH INTENSITY LEVEL. Here, the average of similar time-of-day values is imputed with one minute granularity on different days of the measurement. Following intensity levels are calculated 1mg bins from 1-20mg 5mg bins from 25-100mg 25mg bins from 125-500mg 100mg bins from 500-2000mg
Parameters

- **e (pandas.DataFrame)** – Pandas dataframe of epoch data
- **inputCol (str)** – Column to calculate intensity distribution on
- **summary (dict)** – Output dictionary containing all summary metrics
- **useRecommendedImputation (bool)** – Highly recommended method to impute missing data using data from other days around the same time

Returns

Write dict `<summary>` keys `<inputCol>-ecdf-<level...>mg`

Return type: void

```python
accelerometer.summariseEpoch.check_daylight_savings_crossovers(e, summary)
accelerometer.summariseEpoch.getActivitySummary(epochFile, nonWearFile, summary, activityClassification=True, timeZone='Europe/London', startTime=None, endTime=None, epochPeriod=30, stationaryStd=13, minNonWearDuration=60, mgCutPointMVPA=100, mgCutPointVPA=425, activityModel='activityModels/doherty-may20.tar', intensityDistribution=False, useRecommendedImputation=True, psd=False, fourierFrequency=False, fourierWithAcc=False, m10l5=False, verbose=False)
```

Calculate overall activity summary from `<epochFile>` data

Get overall activity summary from input `<epochFile>`. This is achieved by 1) get interrupt and data error summary vals 2) check if data occurs at a daylight savings crossover 3) calculate wear-time statistics, and write nonWear episodes to file 4) predict activity from features, and add label column 5) calculate imputation values to replace nan PA metric values 6) calculate empirical cumulative distribution function of vector magnitudes 7) derive main movement summaries (overall, weekday/weekend, and hour)

Parameters

- **epochFile (str)** – Input csv.gz file of processed epoch data
- **nonWearFile (str)** – Output filename for non wear .csv.gz episodes
- **summary (dict)** – Output dictionary containing all summary metrics
- **activityClassification (bool)** – Perform machine learning of activity states
- **timeZone (str)** – timezone in country/city format to be used for daylight savings crossover check
- **startTime (datetime)** – Remove data before this time in analysis
- **endTime (datetime)** – Remove data after this time in analysis
- **epochPeriod (int)** – Size of epoch time window (in seconds)
- **stationaryStd (int)** – Threshold (in mg units) for stationary vs not
- **minNonWearDuration (int)** – Minimum duration of nonwear events (minutes)
- **mgCutPointMVPA (int)** – Milli-gravity threshold for moderate intensity activity
- **mgCutPointVPA (int)** – Milli-gravity threshold for vigorous intensity activity
• **activityModel** (*str*) – Input tar model file which contains random forest pickle model, HMM priors/transitions/emissions npy files, and npy file of METS for each activity state

• **intensityDistribution** (*bool*) – Add intensity outputs to dict <summary>

• **useRecommendedImputation** (*bool*) – Highly recommended method to impute missing data using data from other days around the same time

• **verbose** (*bool*) – Print verbose output

**Returns**  Pandas dataframe of activity epoch data

**Return type**  pandas.DataFrame

**Returns**  Activity prediction labels (empty if <activityClassification>=False)

**Return type**  list(*str*)

**Returns**  Write .csv.gz non wear episodes file to <nonWearFile>

**Return type**  void

**Returns**  Movement summary values written to dict <summary>

**Return type**  void

**Example**

```python
>>> import summariseEpoch
>>> summary = {}
>>> epochData, labels = summariseEpoch.getActivitySummary("epoch.csv.gz", "nonWear.csv.gz", summary)
<nonWear file written to "nonWear.csv.gz" and dict "summary" update with outcomes>
```

accelerometer.summariseEpoch.get_interrupts(*e, epochPeriod, summary*)

Identify if there are interrupts in the data recording

**Parameters**

- **e** (*pandas.DataFrame*) – Pandas dataframe of epoch data
- **epochPeriod** (*int*) – Size of epoch time window (in seconds)
- **summary** (*dict*) – Output dictionary containing all summary metrics

**Returns**  Write dict <summary> keys ‘err-interrupts-num’ & ‘errs-interrupt-mins’

**Return type**  void

accelerometer.summariseEpoch.get_wear_time_stats(*e, epochPeriod, maxStd, minDuration, nonWearFile, summary*)

Calculate nonWear time, write episodes to file, and return wear statistics

If daylight savings crossover, update times after time-change by +/- 1hr. Also, if Autumn crossover time, remove last 1hr chunk before time-change.

**Parameters**

- **e** (*pandas.DataFrame*) – Pandas dataframe of epoch data
- **epochPeriod** (*int*) – Size of epoch time window (in seconds)
- **maxStd** (*int*) – Threshold (in mg units) for stationary vs not
- **minDuration** (*int*) – Minimum duration of nonwear events (minutes)
- **nonWearFile** (*str*) – Output filename for non wear .csv.gz episodes
• **summary**(dict) – Output dictionary containing all summary metrics

**Returns** Write dict <summary> keys 'wearTime-numNonWearEpisodes(>1hr)', 'wearTime-overall(days)', 'nonWearTime-overall(days)', 'wearTime-diurnalHrs', 'wearTime-diurnalMins', 'quality-goodWearTime', 'wearTime-<day...>', and 'wearTime-hourOfDay-<hr...>''

**Return type** void

**Returns** Write .csv.gz non wear episodes file to <nonWearFile>

**Return type** void

---

**accelerometer.summariseEpoch.perform_wearTime_imputation**(e, verbose)

Calculate imputation values to replace nan PA metric values

Impute non-wear data segments using the average of similar time-of-day values with one minute granularity on different days of the measurement. This imputation accounts for potential wear time diurnal bias where, for example, if the device was systematically less worn during sleep in an individual, the crude average vector magnitude during wear time would be a biased overestimate of the true average. See https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0169649#sec013

**Parameters**

• **e**(pandas.DataFrame) – Pandas dataframe of epoch data

• **verbose**(bool) – Print verbose output

**Returns** Update DataFrame <e> columns nan values with time-of-day imputation

**Return type** void

**accelerometer.summariseEpoch.writeMovementSummaries**(e, labels, summary, useRecommendedImputation)

Write overall summary stats for each activity type to summary dict

**Parameters**

• **e**(pandas.DataFrame) – Pandas dataframe of epoch data

• **labels**(list(str)) – Activity state labels

• **summary**(dict) – Output dictionary containing all summary metrics

• **useRecommendedImputation**(bool) – Highly recommended method to impute missing data using data from other days around the same time

**Returns** Write dict <summary> keys for each activity type ‘overall-<avg/sd>’, ‘week<day/end>-avg’, ‘<day..>-avg’, ‘hourOfDay-<chr..>-avg’, ‘hourOfWeek<day/end>-<hr..>-avg’

**Return type** void

---

4.3.7 Module contents
CHAPTER 5

Indices and tables

- genindex
- modindex
- search
Bibliography


accelerometer, 36
accelerometer.accClassification, 20
accelerometer.accUtils, 25
accelerometer.circadianRhythms, 29
accelerometer.device, 30
accelerometer.summariseEpoch, 33
<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>accelerometer (module)</td>
</tr>
<tr>
<td></td>
<td>accelerometer.accClassification (module)</td>
</tr>
<tr>
<td></td>
<td>accelerometer.accUtils (module)</td>
</tr>
<tr>
<td></td>
<td>accelerometer.circadianRhythms (module)</td>
</tr>
<tr>
<td></td>
<td>accelerometer.device (module)</td>
</tr>
<tr>
<td></td>
<td>accelerometer.summariseEpoch (module)</td>
</tr>
<tr>
<td></td>
<td>addReferenceLabelsToNewFeatures () (in module accelerometer.accClassification)</td>
</tr>
<tr>
<td>C</td>
<td>calculateECDF () (in module accelerometer.summariseEpoch)</td>
</tr>
<tr>
<td></td>
<td>calculateFourierFreq () (in module accelerometer.circadianRhythms)</td>
</tr>
<tr>
<td></td>
<td>calculateM10L5 () (in module accelerometer.circadianRhythms)</td>
</tr>
<tr>
<td></td>
<td>calculatePSD () (in module accelerometer.circadianRhythms)</td>
</tr>
<tr>
<td></td>
<td>check_daylight_savings_crossovers () (in module accelerometer.summariseEpoch)</td>
</tr>
<tr>
<td></td>
<td>collateJSONfilesToSingleCSV () (in module accelerometer.accUtils)</td>
</tr>
<tr>
<td></td>
<td>createDirIfNotExists () (in module accelerometer.accUtils)</td>
</tr>
<tr>
<td>D</td>
<td>date_parser () (in module accelerometer.accUtils)</td>
</tr>
<tr>
<td></td>
<td>date_strftime () (in module accelerometer.accUtils)</td>
</tr>
<tr>
<td>F</td>
<td>formatNum () (in module accelerometer.accUtils)</td>
</tr>
<tr>
<td>G</td>
<td>get_interrupts () (in module accelerometer.summariseEpoch)</td>
</tr>
<tr>
<td></td>
<td>get_wear_time_stats () (in module accelerometer.summariseEpoch)</td>
</tr>
<tr>
<td></td>
<td>getActivitySummary () (in module accelerometer.summariseEpoch)</td>
</tr>
<tr>
<td></td>
<td>getAxivityDeviceId () (in module accelerometer.device)</td>
</tr>
<tr>
<td></td>
<td>getCalibrationCoefs () (in module accelerometer.device)</td>
</tr>
<tr>
<td></td>
<td>getDeviceId () (in module accelerometer.device)</td>
</tr>
<tr>
<td></td>
<td>getFileFromTar () (in module accelerometer.accClassification)</td>
</tr>
<tr>
<td></td>
<td>getGeneaDeviceId () (in module accelerometer.device)</td>
</tr>
<tr>
<td></td>
<td>getGT3XDeviceId () (in module accelerometer.device)</td>
</tr>
<tr>
<td></td>
<td>getListFromTxtFile () (in module accelerometer.accClassification)</td>
</tr>
<tr>
<td></td>
<td>getOmconvertInfo () (in module accelerometer.device)</td>
</tr>
<tr>
<td>I</td>
<td>identifyUnprocessedFiles () (in module accelerometer.accUtils)</td>
</tr>
<tr>
<td>M</td>
<td>meanCIstr () (in module accelerometer.accUtils)</td>
</tr>
<tr>
<td></td>
<td>meanSDstr () (in module accelerometer.accUtils)</td>
</tr>
<tr>
<td>P</td>
<td>perform_wearTime_imputation () (in module accelerometer.summariseEpoch)</td>
</tr>
<tr>
<td></td>
<td>perParticipantSummaryHTML () (in module accelerometer.accClassification)</td>
</tr>
<tr>
<td></td>
<td>processInputFileToEpoch () (in module accelerometer.device)</td>
</tr>
</tbody>
</table>
S
saveModelsToTar() (in module accelerometer.accClassification), 22
storeCalibrationInformation() (in module accelerometer.device), 32
storeCalibrationParams() (in module accelerometer.device), 33

T
toScreen() (in module accelerometer.accUtils), 27
train_HMM() (in module accelerometer.accClassification), 24
trainClassificationModel() (in module accelerometer.accClassification), 23

U
updateCalibrationCoefs() (in module accelerometer.accUtils), 27

V
viterbi() (in module accelerometer.accClassification), 24

W
wristListToTxtFile() (in module accelerometer.accClassification), 25
writeFilesWithCalibrationCoefs() (in module accelerometer.accUtils), 28
writeMovementSummaries() (in module accelerometer.summariseEpoch), 36
writeStudyAccProcessCmds() (in module accelerometer.accUtils), 28
writeTimeSeries() (in module accelerometer.accUtils), 29