**Prerequisites:**

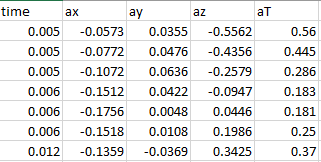
* Python
* OOP (good to know)
* Finite State Machines
* Machine Learning Basics

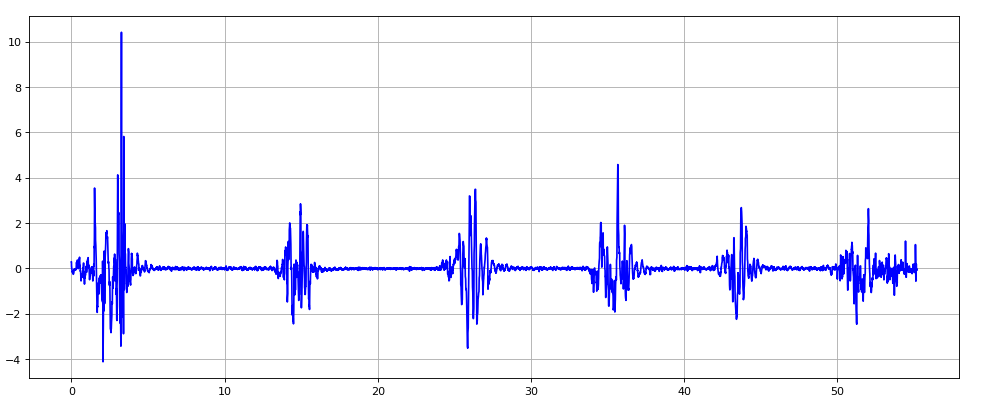
**Activity Recognition using Smartphone Accelerometers**

The activity recognition applications goal is to detect activities (sitting, ,standing and more) being performed by a user who is carrying a smartphone by using measurements of the accelerometer in the phone.

All modern-day smartphones have accelerometers in them. Accelerometers are constantly measuring the acceleration in the x, y and z direction. Various apps exist which can record these acceleration values and store them in a csv file. We have used the "Physics Toolbox Sensor Suite", an Android app, for our measurements. But any other app which records the readings of the accelerometer will do.

Accelerometer readings and the corresponding waveform (along y direction) are shown below:





The goal is to be able to predict with high accuracy, what activity the user is performing from the above sequence of readings.

**Classical Approach**

The classical approach chops up the waveform into small windows and then predicts the activity in that window. For prediction, a predictive machine learning model based on the accelerometer readings is usually used.

The classical approach may work well on vigorous activities such as walking or running for which the accelerometer readings vary significantly depending upon the type of activity. However, for less vigorous activities such as when a person is sitting or standing (and possibly *resting* and doing nothing), it is difficult to distinguish if the person is sitting or standing by evaluating the readings since the readings are not very different if the person is sitting and doing nothing or the person is standing and doing nothing.

**Proposed Approach**

How do we know when a person is *doing-nothing,* if the person is sitting, standing or lying down? When a person goes from a sitting to a standing position, the person needs to *standUp*. Similarly, when a person goes from a standing position to a sitting position, the person needs to *sitDown*. We call **transitional activities** like sitting down or standing as **events**. The idea is then to use events to detect the activity of a person. So, if a *standUp* event is detected, then it is assumed that the person is standing following the *standUp* event until a *sitDown* event is observed, after which the person is assumed to have started the sitting activity. In case a *doing-nothing (or rest)* event is detected when the person is sitting, we know the person is sitting doing-nothing as opposed to say standing and doing-nothing.

The proposed approach is thus based on **events**, and **super** and **internal** **activity** **states**. For example, sitting could be a super activity while *typing* or *doing-nothing* could be internal activities. Note that the same internal activity might occur as part of multiple super activities. For example, *doing-nothing* could be a part of sitting, standing or lying (on a bed).

Every super state has an internal state which we refer to as the default state. In case of a super state like sitting, default state could be doing-nothing. In case of a super state like walking, default state would correspond to walking itself. Hence, if the participant is sitting, doing-nothing, we would refer to the state pair as (sitting, default). If the participant is walking, then instead of referring to the state pair as (walking, walking), we would refer to it as (walking, default).

Events determine the transition between activity states. For example, a sitUp event occurs when a person goes from lying to sitting position. Hence, we can have a transition between activitiy states as (lying, default) -> sitUp -> (sitting, default).

**Implementation**

At the core of the implementation of the activity recognizer are four main components.

1. A finite state machine (FSM), which stores the transitions between the different possible (superState, internalState) pairs based on events.

2. An even stream partitioner which given a sequence of accelerometer readings, partitions the stream into event windows where each window is *expected* to consist of **only one event**.

3. An event classifier which is a predictive model built using machine learning algorithms which can predict an event type from the accelerometer readings corresponding to the event.

4. The activity recognizer. The activity recognizer uses the stream partitioner to partition the accelerometer readings into event windows. It next uses the predictive model to determine the event type from the event windows. Finally, it feeds t ehevent type to the FSM to determine the next (superState, intState).

**1. Finite-State Machine (state\_machine.py, globalVars.py)**

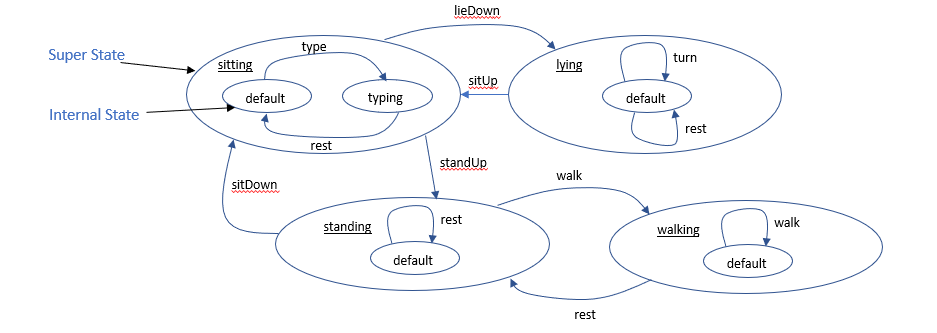
The finite state machine maintains the current state (superState, intState) and given an event, returns the next (superState, intState).

For example, if the current state is (sitting, typing) and the event is standUp, then the state is transitioned to (standing, default).

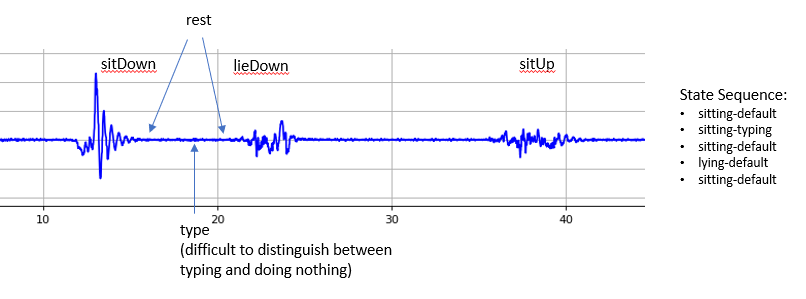
The finite-state machine states and transitions are defined in the file globalVars.py.

The implementation assumes that if for a given state, transition for an event is not defined and the event happens, then the state of the FSM will remain unchanged.

The FSM defined in the current implementation is shown below:



Given a waveform as below, the sequence of states shown next to the waveform will be generated by the FSM.



**The events supported by the FSM determines the events supported by the application.**

**Activities and events Supported (globalVars.py)**

**Super (activity) states**: lying, sitting, standing, walking.

**Internal (activity) states**: default (usually doing-nothing and applies to super states lying, sitting, standing), typing (applies mainly to sitting and standing).

**Events:**

rest – It is an event which is can be described as doing nothing and usually defines a transition from a doing-nothing internal state to the same doing-nothing internal state. But it could also be transitional, for example when a person is walking and then stops, a rest event is said to have taken place.

sitUp - transitional event from lying down state to sitting state

lieDown - sitting to lying down.

standUp - sitting to standing.

sitDown - standing to sitting.

walk - standing to walking.

turn - turning from side to side while lying down, state remains unchanged.

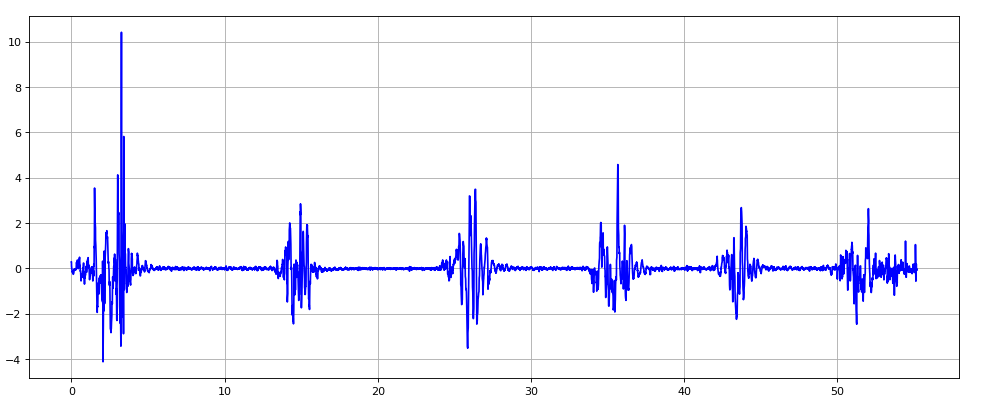
type – expected to occur primarily in the sitting state though it can be generalized to other states.

unk (unknown)

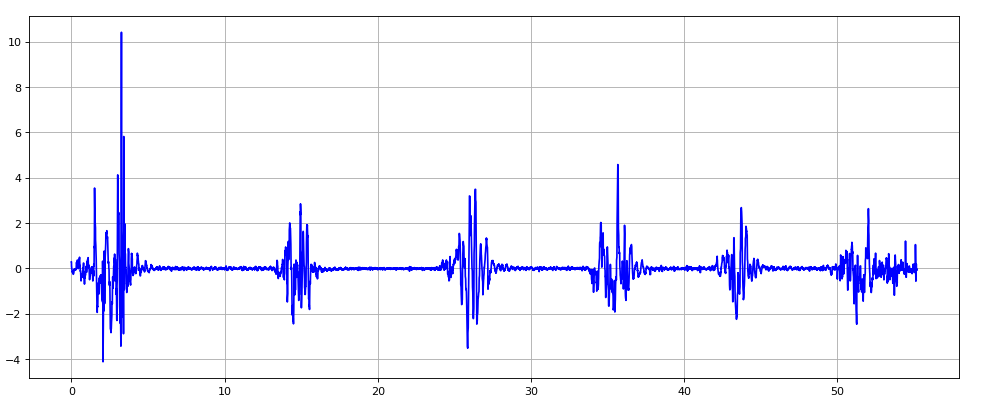
We assume that a set of accelerometer readings from a start to end time is provided. The activity recognition system will use the readings to output a sequence of (superState, intState, initTime, endTime) tuples. The system is broken up into the following components:

**2. Event Stream Partitioner** (**eventStreamPartitioner.py**) – The input is accelerometer readings for possibly multiple events over a certain duration of time. The accelerometer readings have to be partitioned into windows where each window should correspond to an event. The challenge here is that the duration of different events is different. The duration of the same type of event may also differ. We show a correct partitioning of an event stream and a bad partitioning below.

Note that in the good partitioning below, the partitions include a complete event.



Note that in the sub-optimal partitioning below, the partitions are in between the event boundaries and do not include a complete event. Hence, it would be difficult to identify the event from the incomplete set of observations.



The algorithm for event partitioning is experimental. There is significant room for innovation and improvement.

**3. Event Classifier (eventClassifier.py)** – The predictive model is expected to recognize an event, given the accelerometer readings for the event. This model is trained using machine learning techniques. Details on building the model are provided later.

Many features can be generated from the accelerometer readings. A set of features which are giving good accuracy on initial test set have been derived. But there is significant room for innovation in creating other features which provide good accuracy for more demanding test sets.

**4. Activity Recognizer (activityRecognizer.py)**

The activity recognizer is the top-level module. It takes as input a set of accelerometer readings. It generates a sequence of (superState, intState, initTime, endTime).

4a. It initializes the FSM. The initial super state state is assumed as **unknown**.

4b. It reads the accelerometer readings.

4c. It uses the activity partitioner to partition the accelerometer readings into a set of contiguous partitions.

4d. It uses the event classifier to identify the event for each partition.

4e. It feeds each event to the FSM which returns the next (superState, intState). It uses the information to update its list of (superState, intState, initTime, endTime).

4f. After all the events are used up, the sequence of states is returned.

**Building the event classifier – the ML pipeline**

The event classifier is a model which is the output of a machine learning pipeline. To build the ML model, we need to train it with event data. Note that the ML model classifies events. It does not have anything to do with state (super or internal) identification. So, the training data has to be of events, i.e. event features and event labels.

**1. Training data generation**

**1a.** **Record multiple events in a recording session**. The recording session involves a human performing the events such as sitting down, typing, standing up etc. While these events are being performed, the accelerometer app is used to record the accelerometer readings. Each session thus records in a csv file with the accelerometer readings. Manually, the sequence of events is noted for the session.

The start time and the end time of each event is not noted down and will be discovered as explained later.

Multiple sessions are assumed to be recorded. Event types may repeat in a session or across multiple sessions.

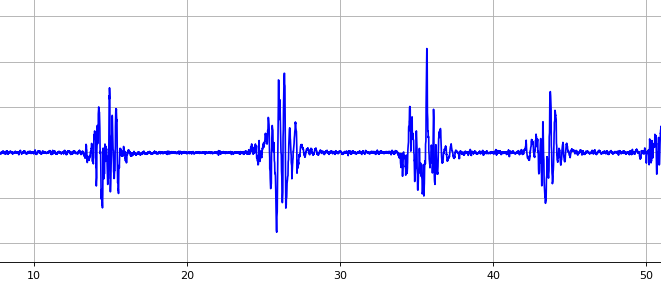
The sessions should generate a sufficient number of events of each type.

**1b.** **Converting session recordings to training data points.** **(trainingSetPrepHelper.py** , eventTrainingData/xyz.csv**,** eventTrainingData/waveformViewer\_xyz.py, eventTrainingData/trainingDataFrom\_xyz.py)**.** Let us say, the session recordings are available in a file xyz.csv. This file may have recordings of multiple events. These raw readings need to be converted into a set of training datapoints where each datapoint corresponds to an event, and consists of features extracted from the data as well as the event label.

Since in a session, multiple events are recorded, we first need to know the start time and end time of each events. Then we can take the raw accelerometer readings between these two times and extract features.

The start time and end time are determined by visual inspection of the session waveform. The session waveform is viewed by executing the code in waveformViewer\_xyz.py. The reason for this manual and visual process is as follows: Different instances of the same event type, for example sitting down, have been found to take different amounts of time and the duration could vary by 1s or more. Moreover, event durations are short, of the order of 1s to 5s. So, recording event start times and end times while performing the recording sessions have been found to be difficult.

Let us assume that we have recorded four events in a recording session, a lyingDown event, two turn events (turning to the left while lying down, turning back to the right) and a sitUp event. Let the waveform (only readings from the y direction shown) for the session be as below. We calculate the start time and the end time of the first sitDown event from the grid to be 13 to 16s by visual inspection. We do the same for the remainder of the events.



**Needless to say, this is a rough and error prone process and improvements are certainly possible.**

Once the event start and end times are known, the accelerometer readings in this time range is used to generate the features for the event. The function used is called convertRawToTrainingInstance and can be found in the file trainingDataFrom\_xyz.py. Each datapoint generated is appended to a dataframe.

In the current implementation, the features 'ZCRY', 'OSCRY', 'minMaxY', 'absAvgY', 'minMaxZ', 'absAvgZ', i.e. 6 values are computed from the raw data for every event, where an event may consist of 100 or 200 accelerometer readings in the x, y and z direction. The calculation of each of these features can be found in the file trainingSetPrepHelper.py.

**There is plenty of room to play around with the feature set, that is new features can be created and used to build models with higher accuracies possibly.**

Thus, trainingDataFrom\_xyz.py, returns a dataframe which consists of the datapoints for events recorded in session xyz.

**1c. Combining training data from multiple recording sessions. (trainingSetPrepHelper.py, createEventTrainingSet.py).**

createEventTrainingSet.py is used to invoke for each session xyz, the corresponding function in trainingDataFrom\_xyz.py. These functions return dataframes which are correspond to events recorded during each session. All these dataframes are concatenated to create the final training data frame called the trainingSet.

**2. Building the model (modelTraining.py)**

The model is trained in the file modelTraining.py. It reads the trainingSet directly from the **createEventTrainingSet.py** file by executing the code in it. The training dataset is used to training the selected model. The trained model, and the scaler used in scaling the data prior to using the data are both stored in a pickle file trainedEventClassifier.sav.

A new model can be created by changing the features used or by changing the machine learning algorithm used for building the model.

**This step needs to be run only if a new model has to be created. Once a model has been created, it can be repeatedly used in the activity recognizer without requiring this step to be rerun.**

**3. Model Evaluation (createEventTestSetAndlEval .py)**

Generation of the data for testing the classifier along with the actual evaluation is done in the file **createEventTestSetAndlEval .py.** The data generated consists of event features and event label, where the event features have been derived from manually specified windows of the accelerometer readings. This step is exactly same as the training data generation.

The features for the series of test events are provided to the model and it is asked to predict the event labels. The predicted labels are compared with the observed labels to compute model accuracy.

**Evaluating the Activity Recognizer(createActivityTestSetAndEval .py**)

Before invoking the event classifier, the accelerometer readings are partitioned into event windows by the **event stream partitioner**, and then the event classifier is invoked on each event window. If the windows are not generated at the right boundaries, we will be asking the event classifier to predict event types which might span part of an event, or could span end of an event and the beginning of the next event. In such cases, we cannot expect the event classifier to generate accurate event types.

A complication to evaluating the stream partitioner is that it uses the event classifier in determining the partitions.

Hence, instead of evaluating the stream partitioner independently, like we do the event classifier, we evaluate the overall activity recognizer application which includes evaluating the event stream partitioner and the event classifier.

The activity recognizer is evaluated in the file **createActivityTestSetAndEval .py**.

The activity recognizer is given a sequence of accelerometer readings and is expected to generate a sequence of (event, starttime, duration, superState, internalState) called the **activitiesEventsSeq** as below, where the superState and internalState together describe the ongoing activity. Initial state is assumed to be “unknown” and initial event as “unk”.

('unk', 0, 11.001, 'unknown', 'default'),

('sitDown', 11.001, 6, 'sitting', 'default'),

('rest', 17.001, 1, 'sitting', 'default'),

('type', 18.001, 1, 'sitting', 'typing')

To evaluate the activity recognizer, we expect the user to provide the two sets of data as explained below in addition to the accelerometer readings.

**Events** **data**. The first dataset to be provided is observations of the events which were part of the accelerometer readings provided to the activity recognizer. It is not expected that all events that occurred during the recording will be provided. It is up to the person evaluating the model to provide the below data which should correlate with the events recorded in the accelerometer readings.

11.0 16.0 sitDown

21.0 25.0 lieDown

35.0 41.0 sitUp

**Activities data**. The second dataset, also expected to be manually provided, is the **activities** data which consists of the major and minor activities observed as below:

11.0 21.0 sitting default

21.0 35.0 lying default

35.0 46.0 sitting default

The evaluation code compares the **activitiesEventsSeq** with the **events data** and the **activities data** to calculate the accuracy of the activity recognizer**.**

Various other debugging output is also generated by the code in the file **createActivityTestSetAndEval .py** to help the developer figure out where the model is making incorrect predictions.

This can be used to tweak the model features, the model itself and/or the event stream partitioner to reduce inaccuracies.

**Miscellaneous sub-components (score.py)**

score.py – this file contains the code for comparing an **activitiesEventsSeq** with thecorresponding **events data** and the **activities data** to calculate the accuracy**.**

**Suggested Activities**

**1. Building the EventClassifier:**

1a. Creating the training data from each recording session.

The accelerometer data is in the folder eventTrainingData/ as csv files. Multiple csv files can exist, one per recording session. For example one.csv is one of them. The sequence of events observed during this recording session has to be noted by the person recording the accelerometer data.

The start times and end times of the events are derived by looking at the waveforms. For every recording, a separate viewer is created. For one.csv, the viewer is waveFormviewer\_one.py.

The features for every event along with the event label are created in the file trainingDataFrom\_one.py.

1b. Creating combined training data.

In the main folder, in the file createEventTrainingSet.py, the training data from the multiple sessions above are pulled into a single dataframe called trainingSet which is written to a csv file, trainingSet.csv.

1c. Building the model. The model is trained in the file modelTraining.py. It reads the trainingSet directly from the above file by executing the code in it. And then the model is trained. The trained model, and the scaler used in scaling the data prior to using it are both stored in a pickle file trainedEventClassifier.sav.

A new model can be created by changing the features used or by changing the machine learning algorithm used for building the model.

This step needs to be run only if a new model has to be created.

**2. Evaluating the event classifier.**

2a. The data for evaluating the event classifier is created exactly as above, but in the folder eventTestData/.

2b. The testdata is read in createEventTestSetAndEval.py. The saved model is also read in. The model is used to predict on the test data and then model accuracy is computed.

If new test data is created and the model performance on the new data is to evaluated, then this step has to be run.

We do not expect the model to be robust on new data since the features and the algorithm is still in experimental stages.

**3. Evaluating the activity recognizer.**

Evaluation of the event classifier requires supplying not only the event details but activity details. The steps required has been explained in an earlier section. The code is in **createActivityTestSetAndEval .py.**