MSc Software Development

Automated personalised music reproducer according to running speed by means of Machine Learning

Andrea Giordano
University of Strathclyde, Glasgow

August 19, 2019

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc of the University of Strathclyde.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself.

Following normal academic conventions, I have made due acknowledgement to the work of others.

I declare that I have sought, and received, ethics approval via the Departmental Ethics Committee as appropriate to my research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to provide copies of the dissertation, at cost, to those who may in the future request a copy of the dissertation for private study or research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to place a copy of the dissertation in a publicly available archive.

(please tick) Yes [✓] No [ ]

I declare that the word count for this dissertation (excluding title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices is

I confirm that I wish this to be assessed as a Type 1 2 3 4 ⑤ Dissertation

Signature:

Date: 19/08/2019

Abstract

In this dissertation, Machine Learning and Neural Network techniques are applied to step recognition and beat detection problem, providing a profound analysis of the full process from the acquisition of the data, to the processing, ending with building and evaluation of the produced models.

It has been particularly examined the potentiality of Long Short Term Memory networks applied to both these context.

The goal of this dissertation is to prove how it is possible to build an Android's application that by making use of the mentioned techniques could reproduce music with the same tempo as the running pace.

After conducting several experiments and testings, the research showed how intriguing results could be achieved on step detection, while further analysis will be needed to beat detection.

It has been finally shown how it is achievable to lively match songs and user's pace in an Android's prototype application, which, in future works, would be possible to combine with the designed and proposed neural network models.

Contents

List of Figures

List of Tables

Preface/Acknowledgements

I would firstly like to thank Dr. Konstantinos Liaskos, lecturer at the University of Strathclyde and my dissertation supervisor for his guidance throughout the year and availability and precision shown during the whole process of my Master's dissertation.

I would also like to express my gratitude to my parents, for the blessing shown to my decisions and the support provided all over these years.
Finally, I would also like to thank Stiven Kulla and Rachel McRae for inspiring and encouraging me through the process of researching and writing this thesis.

Chapter 1

Introduction

This project will provide an in-depth analysis of how it will be possible to improve the actual level of the apps now available in the market to help the user improving them performances during running activities through the reproduction of suitable music.
As it will be shown in the following chapter, various studies proved how during cardio activities listening to music, especially with a similar tempo as the running pace could have multiple beneficial effects.
Having, therefore, an app that can accurately achieve that goal could be beneficial to a wide range of people, from the neophytes to the professionals of the discipline. Indeed reproducing different songs according to the user's pace it could become an entertaining motivational tool to encourage people to have a more active lifestyle, as well as be a powerful means to improve performances of experienced runners.
In this project will be evaluated if it would be possible to overcome, through the application of artificial intelligence techniques, the two main challenges of this project, i.e. detection of the user's steps and identification of the beats in the user's smartphone playlist.
It will be indeed experienced the full cycle of research to produce neural network algorithms, for the definition of methodologies to acquire the database, to the processing of the collected data to the training optimisation and evaluation of the produced model.
The main objective targetted here has been to prove how neural networks, with particular reference to Long Short Term Memory Recurrent Network, could be a suitable

solution to the exposed problems.

Furthermore, it would be intriguing to examine the astonishing abilities of these networks and how, differently to other Machine Learning techniques, can produce exciting results even through the simplification and inaccuracies due to time and resource limitations peculiar to a master's dissertation. Finally, it will be furnished a prototype simulating the behaviour of the proposed app to show and evaluate the potentiality of this project, providing solid foundations for possible future developments.

Chapter 2

Literature Review

Since the usage of Android's build-in sensors to fitness and health application, uniquely integrated with machine learning algorithms, is something not that common and still a topic of research, where a definitive solution yet has to be found, it is understandable how there will not be that much literature available.
The motives of this affirmation are likely to be found in how the availability of these sensors was just heritage of the high-end smartphones until some years ago and furthermore how the accuracy was not high enough, while nowadays even cheap phones have decent quality sensors and are provided to enough computational power to manage similar applications.
Whereas the exponential technological growth is far from over, it is believed that, even if the smartphone's app market is not as flourish as before, there will still be a margin of growth for this niche market as a field whom potentiality is not yet fully explored.

2.1 Effectiveness of music in the increasing of fitness performances

The first question regarding this project to be answered is: if the common perception that music can help performances had been somehow previously scientifically proven.
It is often said that music distracts people from pain and fatigue, elevates their mood, increases endurance, reduces perceived effort and may even promote metabolic effi-

ciency. People run farther, bike longer and swim faster than usual, often without even realising it. [21]

Now would be indeed interesting to investigate the merit of this claim in order to provide more relevance to this work, showing the actual benefit provided.

Some research has been made to prove the correlation between music tempo and sport performances. An intriguing experiment showed the effects produced by increasing or decreasing the actual tempo of a playlist by a 10% during cycling over a self-chosen work-rates program, showing a direct relationship between distance covered/unit time, power and pedal cadence and BPMs. [44]

As this can already create some solid foundation to this case, however, can be furtherly tried to be claimed that movements synchronised with music BPMs could be more efficient. In details has been tested how the value of heart rate (HR) and oxygen consumption (VO_2) change after cycling for 12 min at 70% of maximal HR, resulting in lower (VO_2) in synchronous conditions. Proving that the body has a better efficiency under these circumstances, especially with faster tempo. [8]

Similar experiments have been conducted even on a running sprint over 400-m distance, measuring the influence of motivating and oudeterous (neither motivating nor demotivating) synchronous music. Similar results have been shown, but furthermore emphasizing how is the synchrony more any than other features what boost performances. [25]

An explanation of this can be found in the enhancement in ventral premotor cortex activity when exposed to preferred music tempos [27].

From that we can propose two interesting conjectures, that would be intriguing to investigate in future works. Firstly that, in a moment of effort, this preferred tempo could be very similar to the current running speed, as for the ability of the body to continuously adapt to external inputs. Alternatively, that if not bound by limitations of a pre-ordered playlist, the runner will automatically reach the most performing pace and hence his best potential, as the time of songs follow him.

Having all that in mind it is believed that the most critical feature to sync with the running speed are the song's BPM, and particularly beats with steps, over any other

classifiable musical feature, as genre or triggered emotions.

2.2 Market and Peer-Reviewed analyses

Moving now on to the review of similar applications, a quick market analysis will show how small is the number of available options.
Probably the most popular had been a side premium feature of the well known "Spotify" music app, but that had been removed in early 2018, receiving several complaints as can be seen from their community page [32]. It is hence interesting to see how this type of application is desired by users.
On the market, some other apps try to achieve the same goal, e.g. "Nike+ Run Club" or "RockMyRun". However, they require the connection with "Spotify premium", not available to everybody and just allow you to select a tempo of the played playlist, not to adjust it according to your speed or use the favourite songs. Lastly, some other apps use the GPS sensor in the smartphone to calculate the running pace and after having selected a target speed increase (if you are overcoming it) or decrease (if not achieved) the music tempo. This, lead to a significant issue, i.e. render the app worthless if used on a treadmill, and an inconsistency, i.e. the choice made on how to treat the music to increase motivation.
A better approach has been followed in [36] project, this will be analysed more in-depth as the readiest to be reality wise applied. They propose a solution to both the issues mentioned above. Acquiring data using a redesigned earphone accessory unit, that is including two sensor boards embedded into the earbuds which provide HR data and a baseboard integrated with an IMU (3-axis accelerometer and gyroscope), that collects data from the earbuds and communicates with the smartphone.

The detected HR, combined with the activity revelation, calculated from accelerometer data, provide music recommendations to guide the user to the (previously selected) desired level of activity. This is achieved by reproducing more aggressive songs to increase this level and more chilled ones to lower it. As far as it can be told this is a better means to suggest music compared to the one previously mentioned, considering

Figure 2.1: The Septimu hardware platform [36].

that it would perform in any circumstance and has a more appropriate way to guide to the preferred tempo.

Their approach to features extraction from the raw accelerometer data is to apply the linear magnitude over the three axes:

$$|a_{ear}| = \sqrt{a_x^2 + a_y^2 + a_z^2}.$$

On the resulting samples sequence, it is then calculated the standard deviation x_i with a window of size 50 (corresponding to 1 sec.), and that is the final feature used in the classification process. Then they opted to train an unsupervised learner using a k-means clustering algorithm to identify 3 clusters $\{x_i\}$ corresponding in different levels of activity.

In the real usage phase, each x_i value recorded is classified considering their minimum distance from the cluster heads obtained in training. The activity level is then decided considering a window of 60 seconds by majority voting, in order to detect the right level of activity even though small interruptions as tying shoelaces.

This is an interesting approach and would work very well on a variety of contexts. However, as mentioned earlier, it is the synchronization between the song's BPM with each step what more than anything improve performances. Hence on an athletic environment, a method that precisely detects steps would be more appropriate and would allow a series of more advanced features.

Furthermore, it would be interesting to allow the user to reproduce songs from their owned or favourite music, allowing a more appreciable experience.

Not to mention what could be another barrier for many people, i.e. the necessity of

external devices additional to the smartphone itself that are not available to the mass. The same issue is encountered in [10] or [37], where both use even more external hardware to detect the running pace, as respectively an accelerometer inside an armbar and a heart monitor and 2-axis accelerometer.
Following it will be provided a profound overview of the most common approaches attempted to the step detection and BPM recognition problems, considering them independently in order to supply more accurate and diverse papers.

2.3 Overview of Step Detection's algorithms

In this section it will review the step detection issue, focusing on the most popular type of algorithms, even if used in a different type of application or context, as it would still be possible to be inspired.
Currently, the available literature can be categorized into three main macro-areas: time domain, frequency domain and feature clustering approach.

2.3.1 Time Domain approaches

The most simple usage of the accelerometer sensor is by applying thresholds, above which a step is detected [4]. Nevertheless, though its simplicity of realization and usage, it is challenging to find optimal thresholds, primarily if the smartphone is used in an unconstrained manner.
Another popular approach is detecting peaks of acceleration as in [40] and detecting steps matching them with peaks. The main issue with this is the environmental noise and disturbance that can be misdetected as a step. However, in order to reduce that, different techniques can be applied, as low-pass filtering [46], limiting the time interval between two peaks [14] or use the vertical acceleration [45].
Even though with the last one it could be eventually possible to detect steps even from unconstrained smartphones, their accuracy will still be affected by the user's running speed.
Another option available is zero-crossing counting applied in [28], calculating the num-

ber of zero points in the sensor's data. However, it would also require heavy filtering and work better as a counter than a step detector.
All the methods mentioned till now, greater or lesser extent, still all suffer from a degrade of performances if the smartphone is not firmly attached to the body.
Lastly, it has to be mentioned the autocorrelation approach proposed in [41]. Here, given an acceleration signal, the normalized auto-correlation is computed, producing a new wave with spikes at the correct periodicity of the running activity. This is an attractive idea and something that could have also been applied in this project because of its great performances at a reasonable computational cost.

2.3.2 Frequency Domain approaches

This type of techniques have a different approach, focusing on the frequency of signal measured over successive windows, rather than time.
Different type of algorithms can be applied to transform the signal from its original time domain to a representation of its frequency.
The most commonly used are "fast Fourier transform"(FFT) [17] or "short-time Fourier transform"(STFT) [9]. The main difference between the two is that STFT determine the sinusoidal frequency and phase content on shorter segments of equal length instead that over the whole signal(FFT), resulting more meaningful.
Another group of algorithms are the "continuous/discrete wavelet transforms"(CWT/DWT) [9] [43]. Here is explored the idea of transforming the wavelets of a signal in order to provide an overcomplete representation of the signal itself. The difference between the two approaches is how translation and scale parameter of the wavelets are idenitified: varying continuously in CWT, or predefined value in DWT.
These type of algorithms could be even used to preprocess the data before applying one of the techniques seen in the previous section, or that will be illustrated in the next one.
The main issue encountered with these approaches is that Android's accelerometer provides a new data point when it is detected a new value by the sensor, hence the interval between these samples is not regular, so they cannot be treated as a constant

signal. As far as it has been possible to experiment, at the application level, the only way to have a constant rate is to discard data point received under the selected rate. However, it adds computational power to an operation that, as it will be in deep explained later, is already problematic. Not to mention that thinking of the working app, some of the missing values could be the actuals steps, resulting in weak predictions. Alternatively, it would be possible to modify the Android framework internals (sensor HAL), but that has been considered out of the researcher area of expertise.

2.3.3 Feature Clustering approaches

These kinds of approaches take advantage of machine learning techniques, and they have been mostly used in medic and rehabilitation contexts. Several types of algorithms had been applied over the years to classify the gait phases, starting from state machines [38] on angular velocity and force-sensitive resistors, to Hidden Markov models(HMMs) [30] on signals collected with a mono-axial gyroscope. Not to mention KMeans clustering algorithms as previously mentioned in [36].

Nevertheless, even if not entirely continuously sampled, sensors data can be considered a time series. Hence it would make sense to apply a time-recursive neural network(RNN), because of its better decision-making ability, as it considers not only the present but also the recent past.

This intuition has been proven to be a valuable starting point by previous work such as [47] where the preprocessed values of rotation rate and acceleration over the three axes are used as input for a two-layer Long Short Term Memory Neural Network to count steps.

To conclude this section, it will be finally mentioned [13] for the extensive comparison of various walking detection and step counting approaches. It has indeed been highlighted how for the former the best results are obtained using thresholding based on the standard deviation and signal energy and for the latter windowed peak detection, HMM and CWT. [24] Their work is giving more credit to the hypothesis that an RNN and particularly an LSTM one could lead to optimal results since it is a more advanced

version of an HMM.

2.4 Overview of Audio Beat Tracking and Music Tempo Extraction's algorithms

Having already made a point of why to prefer music tempo over genre classification in this context, the next step is to understand what could be more appropriate between Beat Tracking and Tempo Extraction. As the second one could be an easier task since it does not require the knowledge of every single beat, it is although more sensible to errors and results misleading in the case of songs with variable tempo.
With that in mind, we can state that beat tracking is a better option for music synchronization with external elements, as it has to be in this application.
Hence considering that the primary goal of this project, it will hence furnish two similar topics of investigation. Since the audio signal and, to some extent, values from accelerometer and gyroscope sensors both have the form of a wave and in the prediction process, previous values will be relevant in the decision-making. However, beat tracking and tempo estimation had been usually treated simultaneously, often subdividing the overall problem in two phases: "a first stage that generates a driving function from direct processing of the audio signal; and a second stage that detects periodicities in this driving function to arrive at estimates of tempo and/or beat times" [31].
This scheme is applied by most of the traditional algorithms, using first a filter bank [3] or the discrete Fourier transform(DFT) [29] [22] and then pitch and onset detection methods [3] or banks of oscillators [22].
Nevertheless, in recent years, new approaches have been displayed, and most of them involve NN like CNN or RNN. The former had been used mostly for genre or artist classification [16], but found its way even on Onset Detection [42], showing promising performances. However, the same researcher, S. Böck, has then provided optimal results even on the beat tracking problem participating to several of the lasts "Music Information Retrieval Evaluation eXchange" (MIREX) [33], a competition that awards several music classification tasks, including Beat Tracking. As an example, in [11] an

artificial neural network composed of three LSTM bidirectional layers with a final softmax activation to make predictions in the range [0,1] is fitted with Mel spectrogram of sequences of length 23.2 ms, 46.4 ms, and 92.8 m calculated with the STFT.
The just mentioned topics will not be deeply covered here in the belief that it would be possible to explain them in a much precise way providing, in the following chapters, the theory alongside the practical example of how it has been applied.
Concluding, having a look to the MIREX's best placements over these last years, it is observable how neural networks are a valuable and widely used methodology to approach this topic. This can also be evinced by the evaluation of different deep neural network(DNN) over this task provided by [23].

2.5 Objectives

With the provided overview of how Historically similar problems have been treated, it is possible now to state clearly the project objectives:

- Examination of how could be achieved a step detector algorithm relying on Android's smartphone sensors by the use of ML techniques and LSTM neural networks;
- Examination of how it would be possible to obtain a beat detector algorithm by mean of Neural networks;
- Implementation of an Android app able to perform the proposed behaviour, i.e. matching the runner pace with the BPM of the closest song available.

Chapter 3

Methodology

3.1 Requirement

Since this, has been a self-proposed project, it was not immediate to gather requirements, as there was not anything like a "customer's request" to be satisfied or other particular guidelines to direct the continuation of the project.
Therefore, already having to define the outline of this in the individual project of the module CS 993, it has been useful to start an early requirement's gathering from some potential users, to have a better understanding of which could be the essential features.

3.1.1 Requirement's Gathering

It has been opted for an online survey, with mostly closed question as it seemed the less time and effort consuming option for potential candidates. This is because it is understandable how people can be busy and can quickly lose interest in open-ended questions, hence skipping them or replying poorly. That had been even proven by the low level of feedback received over the three open questions proposed, with just two replies in one of them and none the remaining two.
Having in mind the form of it, it has been then targeted a diverse group of people, restricted on the range between 20es and 30es because these seemed to be potentially the

most interested customers Fig. A.1. It has been assured to cover the broadest range of potential users, from sports fanatics, to occasional runners, to not sporty people Fig. A.3.

Something that has been then particularly interesting was to have a confirmation that the building effort could be potentially worth it, as the assumption that the final product could be desirable could have been somehow supported by numbers.

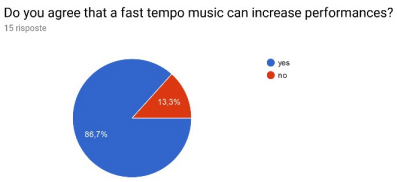


Figure 3.1: Effects of Fast Tempo Music

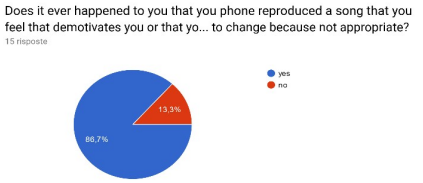


Figure 3.2: Undesirable Effects of Inappropriate Music

This has indeed been tested with three different questions: if respondents actually listen to music while running Fig. A.4, if they ever felt demotivated by the next song reproduced from the playlist Fig. 3.2 and if they could appreciate this kind of app Fig. A.7. The results had been quite promising with respectively 73%, 87% and 100% of positive answers.

Moreover, It has been interesting to see how widespread was the perception that music, particularly fast tempo once, could improve performances Fig. A.6 Fig. 3.1 and additionally to understand how users would favour to be motivated and their preferences on how to handle the transition between songs and different tempos.

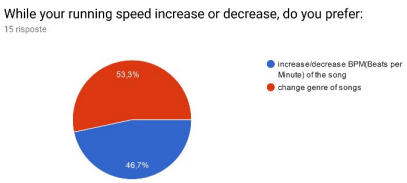


Figure 3.3: Preferred Music Feature to be Adjusted

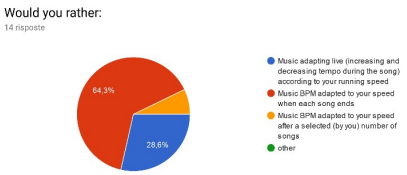


Figure 3.4: Preferred type of Tempo Adjustment

It turns out that firstly, a slight majority of them would prefer a change in the genre more than in bpm Fig. 3.3, against the researcher initial belief and this was something that pushed to carry out more careful and in-deep analysis as read in the dedicated

section of the literature review. Therefore, considering that question's results were far away from a vast majority and opposed to the proofs provided by the listed peer-reviewed articles, it has been decided to follow the BPM approach.
Secondly, the most requested type of tempo adjustment has been after each song with an impressive 73%, followed by the real-time adaptation of the BPM at song level with a 29% Fig. 3.4. This majority is something that makes sense, as it allows a certain amount of flexibility. Since when the desired speed is achieved, it could be possible to run at steady velocity using the music beats as a reference to maintain the right pace, always having an appropriate song. Or if it is preferred training with different workloads, songs after songs the music will follow with just a small delay.
These are the motivation that droved to this choice for the actual project. However, it would be still interesting in future works to make some test to see if it is doable an individual beat-step alignment and how that would impact on user performances.
Another issue that has to be solved was to select an appropriate phone placement for this experiment and, as can be evinced from Fig. 3.5, the most commonplace is the pocket. This is one of the most relevant info extracted from this survey as over this revelation it has been based the assumption that the smartphone will be held on the pocket and that, as it will be subsequently shown will influence all phases of the project. Undoubtedly, this is a potential restriction to the potential usage of the app, but is something required by the strict time available, as it would exponentially increase the project complexity.
Lastly, respondents showed interest in performance tracking, as shown in Fig. A.8 and Fig. A.10 and particularly in the possibility to reproduce more often songs that lead to better results Fig. A.9. Nevertheless, that would have been a shareable secondary feature, unfortunately this will be another thing to explore in future works.

3.1.2 Functional Requirements

Having now a better picture of the project and his main challenges, it has been possible to proceed on the next steps to extract the Functional Requirements, useful to build a working plan for the succeeding weeks.

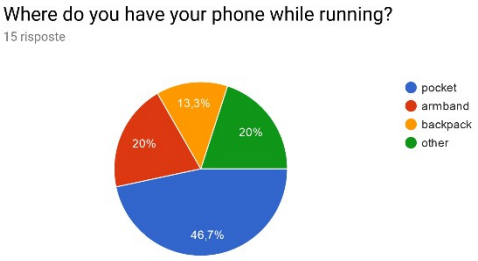


Figure 3.5: Preferred type of Tempo Adjustment

Following Agile suggested principles, it has been decided to produce some **User Stories**, with the *role-goal-benefit* rule, to have a better idea of the requirements from a user perspective point of view. To then categorize them using the *MoSCoW* method to organize the tasks in a user perspective way, estimating the potential effort and design constraints.

Particularly challenging had been producing small and independent *user stories*, because from a user perspective the actions to perform are quite limited, with most of the functionality happening on the back-end and therefore cannot be adequately expressed as user stories. It is indeed one of the goals to reduce the number of actions required as much as possible to leave user freedom to concentrate on running.

Below are the main user stories produced with relative estimated effort:

- *As a user, I MUST decide if give the app permission to read on my device storage so that can access and reproduce songs; 6/10*
- *As a user, I MUST be able to see the detected songs in the smartphone internal or external storage, to check if they are detected and be able to select them; 6/10*
- *As a user, I MUST be able to decide if I want to start the music reproduction with a song selected by me or the closest to my steps per minute; 7/10*
- *As a user, I MUST be able to select the desired song, to start the reproduction of it; 7/10*
- *As a user, I MUST be able to touch the play button, so the music from my phone will start accompanying me during the running session; 8/10*

- *As a user, I MUST be able to touch the stop button to stop the reproduction of music; 7/10*
- *As a user, I SHOULD be able to decide if I allow vocal notification so that I can or not be notified about my running speed; 7/10*
- *As a user, I COULD be able to remove a song I did not like from the playlist; 6/10*
- *As a user, I COULD check my previous performances to see my improvements; 8/10*
- *As a user, I COULD be able to see which songs lead me to the best performances so that I can know what works better; 9/10*
- *As a user, I COULD listen more often to the songs that lead me to the best performances so that these can increase; 9/10*

Over these, it is then possible to build some **User Case Description** in order to subsegment even more complex issues to make them more understandable and ready to be implemented. It will now be proposed an example of probably the most crucial function o this system.

Use Case Name:	UC1: Start Music Reproduction
Triggering Event:	The user press the play button
Scenario Description:	The user wants to listening to appropriate music without having to worry to have to change them
Main Actor(s):	User
Preconditions:	1. User allowed the app to access storage; 2. App correctly imported songs; 3. App have BPM for each song; 4. App is successfully detecting and counting steps;
Post condition(s):	User can listen to appropriate songs without having to change them
Main Path:	1. User open app; 2. User press play button; 3. User start running;

Table 3.1: User Case: Start Music Reproduction

These are the requirements that have been initially prefixed, and over these, the project has been designed.

However, being a goal to experiment and elaborate the full process of the developing of ML algorithms, to better understand the concerning challenges and where it could be possible to adopt improvements.

The trade of for it was having to give up on the possibility to realise a final product, opting for just a prototype, although with the advantage of offer a better overall picture of the subject matter.

This choice is indeed extremely reflected in the decision of not using the Android built-in step counter, but rather try to implement a custom one.

As a result, it has been needed a second app to gather the data from the smartphone's

sensors to build a dataset big enough to then train the ML algorithm. In order to achieve that it is essential to feed it with data point appropriately labelled as *step* or *no-step*.

Here another assumption has been made: the researcher has to be the only tester during the whole process. It can be understood how, particularly during the collection of data for training purposes, accuracy is crucial and how this will be projected on prediction's precision. As far as it has been possible to experiment, considering how Android allows accessing sensors, there was two main option to label the sensors' data, relying on the build-in step detector or on user inputs when a step is performed. However, considering how the phone has to be carried in the pocket from starting of recording and any manual touch could compromise the integrity of data. Having performed some tests, the best option found had been to use the headphone media button to manually record each step. This will allow a better accuracy of recording compared to the Android's built-in detector, and moreover, it does not require any physical contact with the phone.

Figure 3.6: Android-Manual Detected Steps over Z axis Accelerometer (Start of the session)

Figure 3.7: Android-Manual Detected Steps over Z axis Accelerometer (End of the session)

As can be seen in A.11 and particularly observing the start Fig. 3.6 and end Fig. 3.7 of a sensor's recording session, Android's detector miss some initial steps at the beginning and identifies some extra ones at the end. Moreover, it is visible how the precision of its detection is even less accurate than manual ones.
Over these considerations, it is then possible to build some **user stories** to guide in the building phase:

- *As a user, I MUST decide if give the app permission to read and write on my device storage so that can write sensor's data; 7/10*
- *As a user, I MUST be able to read clear instruction on the app's usage; 1/10*
- *As a user, I MUST be able to touch the play button, so that the app starts recording sensors data; 9/10*
- *As a user, I MUST be able to touch the stop button, so stop recording data; 7/10*
- *As a user, I MUST be able, after having stopped the recording, to send data in a usable format; 6/10*

Finally, once having this app and then an appropriate dataset that the next step is to implement the two ML algorithms to detect steps and song's beats. These are the other two *functional requirements*.

3.1.3 Non-Functional Requirements

Considering indeed the two ML algorithms, it is evident how they are susceptible to some Non-Functional Requirements as *performances* and the main method of evaluation will be *accuracy*, *binary cross-entropy log loss* and *precision*.
Thinking instead to the final prototype and its possible future development are relevant principles like *availability*, *serviceability*, *reliability*, *maintainability*, *manageability* and *usability*. These are all characteristics that aim to assure a smooth, bug-free and enjoyable experience, a crucial point in this system as its success will heavily depend on user satisfaction. Having instead in mind the app to gather data, another relevant principle followed has been *data integrity*, as utterly useless if somehow corrupted during the recording, writing or collecting phase. Not to mention that even here some special attention was given on *performances*, since as will be explained, record and write multiple sensors at the same time could be a challenging operation.

3.2 Methodologies

In order to support and help the organization of the process of software development many theory had been developed from the more traditional *Waterfall*, to the more modern *Scrum*, *Extreme Programming*, *Rapid Application Development*, *Spiral* and *Agile*. The overall idea followed here has been to draw lessons from all of them, trying to recreate a tailored strategy that could fit the specifics needs of the project.
Certainly *Waterfall*, even though been overtaken by newer and more dynamic methodologies, still provide, with its *Requirements-Design-Implementation-Verification-Maintenance* structure, an useful tool to start decomposing a project in more approachable problems and force to adopt a methodical approach.
However, the better flexibility provided by *Agile* methodologies are welcomed as being a customer-oriented and experimental project it has been immediately clear how the use of *user stories* or *CRC cards* were crucial facilitators. In the same way, it has been adopted a *Test and Goal-Driven Development* approach, performing constant testing and alongside continuously adjust requirements and plan for the following weeks.

In a similar context, is where *Scrum* methodology, especially with *Sprints* are particularly valuable, forcing to produce every week some new functionality to test and evaluate, keeping high the level of motivation and providing a better time awareness.
Another reason that driven to opt for this technique is that had been clear since the early stages how, be able to create and stick with a predefined plan, was almost impossible. The reason for this is how being these topics yet fertile ground for research and especially new to the researcher it had been easy to underestimate challenges and therefore has been extremely rare the case that estimated effort was appropriate.
Therefore, considering meetings with the supervisor a sort of *expert evaluation* the organization of the project resulted in weekly *sprints* where results where evaluated and from that the following steps where decided.

3.3 Programming Languages and frameworks

In the choice of the preferred programming language some reflection has been made, starting with the knowledge that will be a mobile application, thus becomes particularly relevant the decision of targetted operating system.
The main available choices here are Android, IOs or eventually both in the case of an hybrid application.
Therefore considering the latest data regarding market shares [1] the two main actors have respectfully 76% and 22%, it becomes then quite clear how Android is a must have and therefore, considering the time available,the idea of a native Apple application is discarded.
However there i still the possibility to cover almost the totality of the market with an Hybrid application, this is possible thanks to the *React Native* framework that allow to writing natively rendering mobile applications, with the advantage that being based on *React* (a very popular *JavaScript* library) most of the code is reusable on both platform.
Nevertheless in this particular project it could be not the optimal choice, as it rely heavily on accelerometer and gyroscope sensors and to access them it is always recommended to write native code. Not to mention the advantage of using *Android*

Studio with its Grade Build System, that allows to include external binaries or library modules to its build as dependencies.
Secondarily has also been considered the amount of time needed to mastery a new language like JavaScript and the less control over variables that this offer to the programmer compared to Java.
Being therefore native the best option available, then has been preferred *Java* over *Kotlin* as still the most popular option out there and *Android Studio* as preferred *integrated development environment (IDE)*, since is the official one and being build over JetBrains' IntelliJ IDEA it allows lots of functional keyboard shortcuts and to efficient autocomplete suggestions.
Furthermore an Android external library named *Dexter by Karumi* [26] has been used to request user's read permission at runtime with the advantage to simplify the process allowing to initialize objects or call methods only if the permission has been granted.

A separate discussion then is required by the developing of ML algorithms, indeed the data has been acquired by a developed android application build in the previously mentioned method, but once obtained all the process of feature extraction and model development had been carried on in *Python* language.
The main reason behind this choice is the extended availability of libraries and framework that make easier different phases of the process. Here is a list of them:

- *Pandas* library for data manipulation and analysis, particularly useful in this case to read CSV files and especially for his data-frame management utilities like grouping, splitting, adding or removing rows and columns and finally its time series-functionality like moving window statistics.

- *NumPy* library supporting multi-dimensional arrays and matrices, allowing different type of transformation over them, alongside a wide collection of high-level mathematical functions like the very useful here *FFT*.

- *Matplotlib* library allow production of different types of plots, crucial to exanimate data and evaluate results.

- *Scikit-learn* library implementing various classification, regression and clustering algorithms including support vector machines, random forests and k-means, plus many usefull preprocessing functionalities like scaling or weighting data.
- *SciPy* library specialized in scientific and technical computing, conteining specific module for signal processing techniques like *mfcc* and *logfbank*.
- *LibROSA* library for music and audio analysis, facilitating some fundamentals operation like loading audio files, compute spectrograms and locating beats.
- *TensorFlow* library used for ML applications such as NN, furthermore allows deployment of computation across *CPU* and *GPU*, thanks to *CUDA* extension, sensibly reducing operation's time.
- *Keras* library running on top of TensorFlow, enabling to build NN models in a faster and user friendly way, with particular regard of layers, activation functions and optimizers.
- *FFmpeg* a free and open-source software suite of libraries and programs for handling video, audio, and other multimedia files.
- *Traktor* a DJ software developed by **Native Instruments**

3.4 Design

This section will now focus on the followed process to define the architecture components, interfaces, and other characteristics of this system. In its development, it has been particularly useful to always have in mind some suggested principles to produce a more appropriate design:

- *Abstraction*, focusing only on the relevant information of each object analysed, ignoring the remaining;
- *Coupling and Cohesion*, ensuring the interdependence of modules and the level of association on the elements of each of them;

- *Decomposition and modularization*, highlighting the importance of dividing the system into smaller components and interfaces;
- *Encapsulation and information hiding*, highlighting the importance of grouping and make inaccessible internal details;
- *Separation of interface and implementation*, focusing on having public interfaces separated from the detail of how the component would be realised;
- *Sufficiency, completeness*, and primitiveness ensuring that each component capture only the important characteristic of abstraction and that the design will be easy to implement;
- *Separation of concerns*, allowing stakeholders to focus on few things of their interest.

Now, it has to be reminded how the design of the prototype can not be considerate completely finalised and is something that could be altered when the implemented ML algorithms will be integrated to find optimal performances and user experience.

3.4.1 High-Level Design

The first useful step to start transforming the gathered requirements into high-level decisions is to utilize CRC(Class-Responsibility-Collaboration) Cards.
What seems to be particularly helpful is that producing them helps to start thinking about the high and low-level design, providing a physical visualization, avoiding to be stuck in abstraction and helping to create a better plan to follow.
As can be seen in an example of CRC card produced Tab. 3.2, responsibilities of a class are a rough oversimplification of what will become a method and this help start thinking early of how they will look like, forcing the production of cleaner and more organized code in line with Agile methodologies.

STEP-SONG MATCHER	
• Get step count; • Get song Collection; • Read songs' BPM; • Find the song with closest BPM to step count; • Return the more appropriate song;	• Step Counter; • Song Collection; • Song; • Music player;

Table 3.2: Step Counter CRC card

Furthermore, this has been the phase where important decisions have to be taken. Indeed some assumption has been again required. Considering how it will be just a prototype and will not yet implementing the investigate ML algorithms, some simplifications have been adopted.

Firstly, it will be used the Android's built-in step counter, that, however, does not change the behaviour of the application majorly.

The situation is instead different regarding the BPM detector. Here it is assumed that the device will contain audio files with as a name their BPMs value. Hopefully, in possible future works, the method that accesses the file name will be replaced with a model that performs the operation independently, but as now it would require some manual work.

3.4.1.1 Architecture and Patterns Design

An interesting approach, helpful to brainstorm over a complex problem and built fundaments to the high-level design, is the old *main program—subroutine architectural pattern* by Niklaus Wirth paper [35], where he for the first time formally defined the top-down problem decomposition methodology, and that led to the formal definition of this pattern [20].

It consists of the continuous decomposition of the whole idea in sub-problems and so

on, allowing to deeply think about how each functionality is composed. Therefore this is still a valuable tool that, if used with due proportion, allows to achieve a better *decomposition and modularization*. In the following Fig. 3.8, it is visible how all the system functionality finds a graphical representation, improving class organization and design.

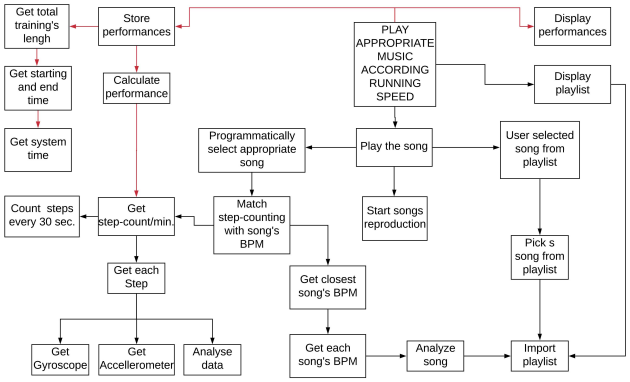


Figure 3.8: Main program-subroutine pattern

It can also be seen how using red arrows are also taken into consideration user's performance collection and display functionality that could be a starting point for future extensions of this project.

Nevertheless, this is an Android application, and as mentioned in the previous section it has been used *Java*, that is strictly an *Object Oriented Programming Language*, therefore will require a dedicated architectural pattern.

Probably the most popular one is the *Model-View-Controller (MVC)*, it "was originally developed to map the traditional input, processing and output roles of many programs into the GUI realm" [20].

What is particularly interesting is how it suggest to segment an application or interface into three parts, increasing modularization and reusability of the code:

- the *model*, responsible for managing some data elements, responds to queries about its state, and to instructions to change it,
- the *view*, in charge of managing the display area and responsible for presenting

data to the user through graphics and text,

- the *controller*, in control of interpreting inputs from the user and translates them into commands sent to the model and/or view to effect the appropriate changes.

It is indeed beneficial in order to achieve *separation of concerns* and *separation of interface and implementation*, braking or program in how it works, what it is displayed and how it gets the data.
Similarly, as can be seen in Fig. 3.9 helps in *encapsulation and information hiding*, in fact, leads to the creation of different classes and packages, these will produce objects that hide data and how they are manipulated, presenting to the world just a simple interface to interact with.

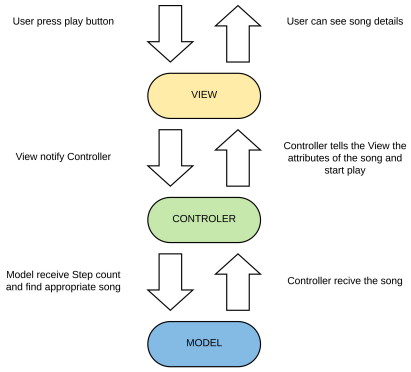


Figure 3.9: MVC Pattern

3.4.1.2 User Interface Design

One of the last steps in the high-level design is the development of the Graphical User Interface (GUI), that, in a leisure-fitness context, could be a relevant factor for the succeeding of an app. Although this is just a prototype to show the potentiality and give a flavour of the final behaviour of this app, it has been developed accordingly the **10 Usability Heuristics principles**, with particular attention to two of them:

1. *Aesthetic and minimalist design*, using a completely black background, hiding Android toolbar and navigation bar, principally for aesthetic reasons, having a more immersive experience, but it also has the advantage to be less battery

draining, considering how much is already stressed by step detection and song matching.

2. *Recognition rather than recall*, reducing as possible the number of actions and buttons required by the user, avoiding them to have to remind long sequence of actions, distracting them from what is the real goal of the app.

Moreover, considering how the device during activity should be placed in the user's pocket, hence it is not really possible to have access to the screen, therefore, to achieve *visibility of system status* vocal notification had been added to advise the user over his running speed. This functionality can also be turned off simply touching a button, as this could be a motivational tool, but it is important to leave the user the final decision.

Figure 3.10: RuntoBPM Main Activity

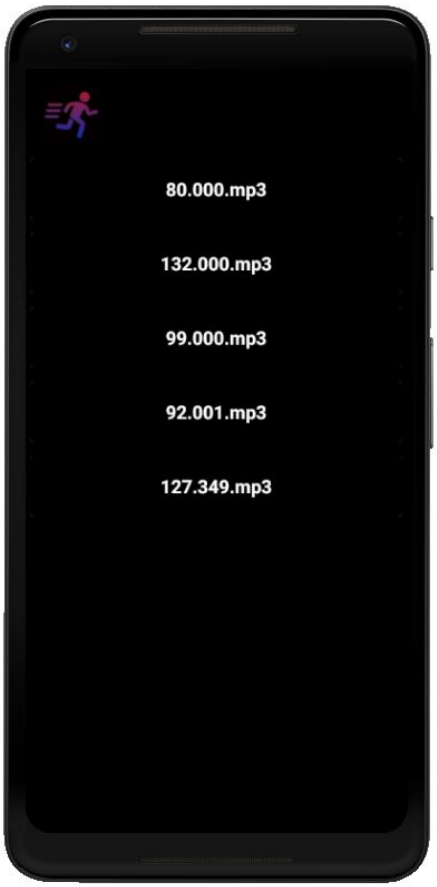


Figure 3.11: RuntoBPM Song Collection Displayer Activity

With all that in mind, a brief overview of the main functionality will be provided.

As can be seen in Fig. 3.10, it is always displayed the current running speed, and three more buttons are present:

- The *Play button*, that starts the music reproduction and will make appear below it the name of the played song; it will be always reproduced the closest song to the user's pace, in the form of steps per minute; when they will start running, a slow-paced song is played to follow their warm-up, and as they increase the speed, song after song will follow their rhythm.
- The *VoiceOn button* start or stop the vocal notifications.
- The *Collection button* leads the user to another screen where all the available songs are displayed.

The second available screen Fig. 3.11 will also allow starting the reproduction of the desired song clicking on it. This will however still provide an automatic selection of the following songs with the already mentioned logic. Moreover, the button located on the top left will return to the main activity.
All these functionalities are quite simple and would not require further instruction; nevertheless, a brief description could be provided alongside the deployment.

3.4.2 Low-Level Design

With clear in mind, the main decided functionality will be now possible to dig deeper into *Low-Level Design* details. The *UML diagram* in Fig. 3.12 Will provide an overview of how classes have been structured and which methods have been implemented.
It is possible to observe how in order to be able to access and therefore display and play songs both in *MainActivity* and in *CollectionDysplayer* activity, it has been implemented an Application class named *RunToBPM* (name of the app) extending `android.app.Application`. In doing so, it will be possible to call its methods initializing that in both activities.
This is something required, not only to have cleaner code, avoiding duplications but above all to initialize there the `MediaPlayer` and `SongCollectionManager` classes and therefore not having to pass these objects between activities through `intents` or

`bundles`, resulting in a very cumbersome and not bug-free procedure.

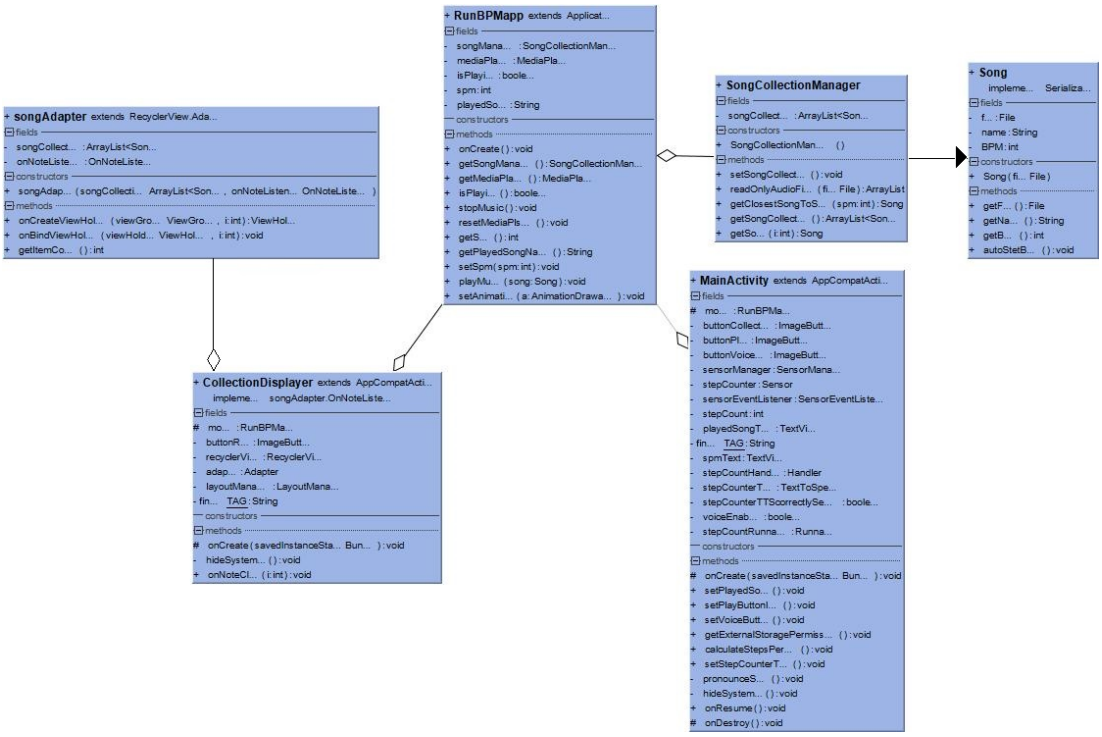
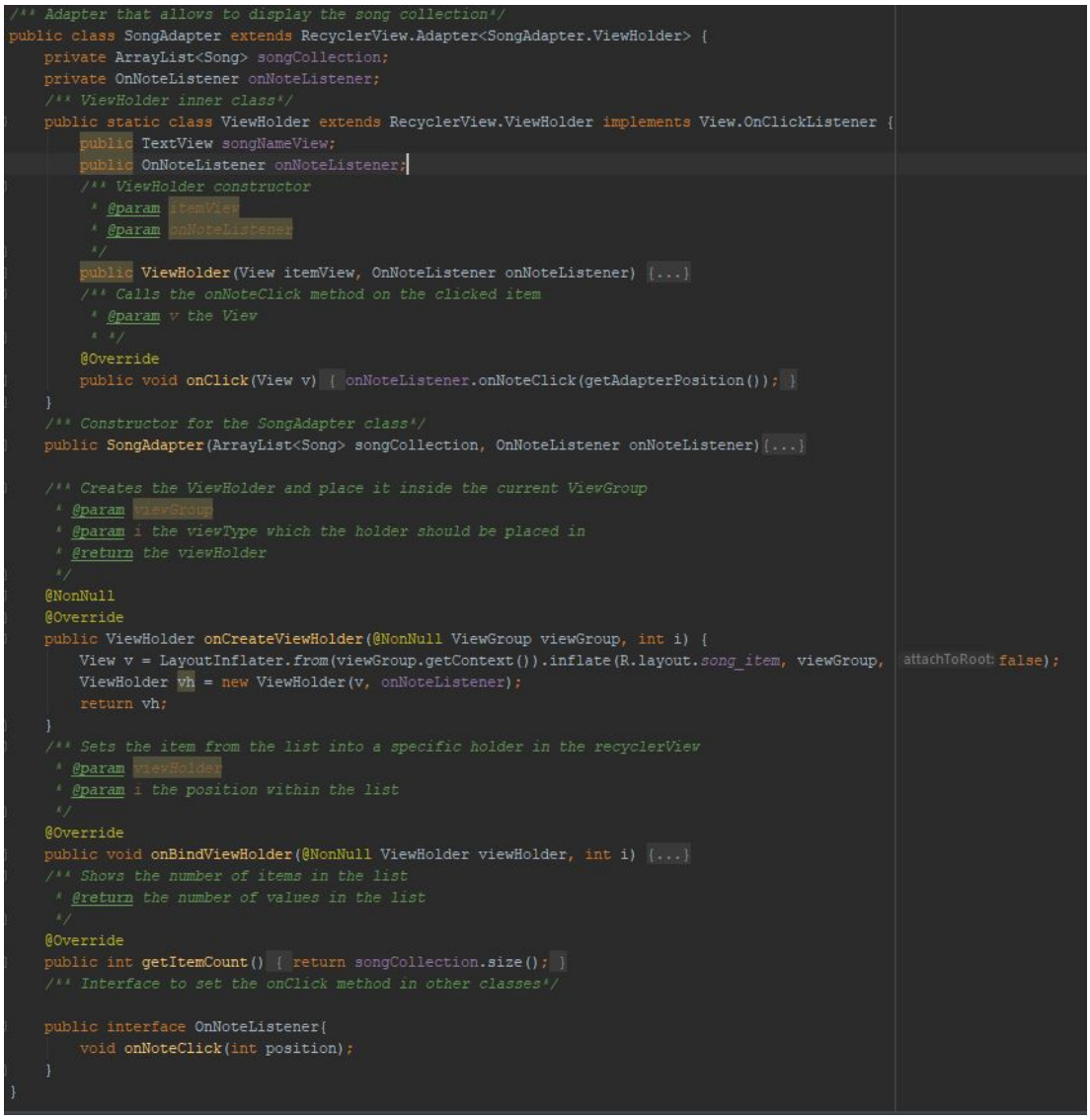


Figure 3.12: RunToBPM UML Diagram

It has also been used the interface `onNoteClick`, noticeable in Fig. 3.13 in to define what will happen when a song is clicked, since the behaviour of the `RecyclerView`, assigned to display them, is handled by a created custom adapter named `SongAdapter`.



```
/** Adapter that allows to display the song collection*/
public class SongAdapter extends RecyclerView.Adapter<SongAdapter.ViewHolder> {
    private ArrayList<Song> songCollection;
    private OnNoteListener onNoteListener;
    /** ViewHolder inner class*/
    public static class ViewHolder extends RecyclerView.ViewHolder implements View.OnClickListener {
        public TextView songNameView;
        public OnNoteListener onNoteListener;
        /** ViewHolder constructor
         * @param itemView
         * @param onNoteListener
         */
        public ViewHolder(View itemView, OnNoteListener onNoteListener) {...}
        /** Calls the onNoteClick method on the clicked item
         * @param v the View
         * */
        @Override
        public void onClick(View v) { onNoteListener.onNoteClick(getAdapterPosition()); }
    }
    /** Constructor for the SongAdapter class*/
    public SongAdapter(ArrayList<Song> songCollection, OnNoteListener onNoteListener){...}

    /** Creates the ViewHolder and place it inside the current ViewGroup
     * @param viewGroup
     * @param i the viewType which the holder should be placed in
     * @return the viewHolder
     */
    @NonNull
    @Override
    public ViewHolder onCreateViewHolder(@NonNull ViewGroup viewGroup, int i) {
        View v = LayoutInflater.from(viewGroup.getContext()).inflate(R.layout.song_item, viewGroup, attachToRoot: false);
        ViewHolder vh = new ViewHolder(v, onNoteListener);
        return vh;
    }
    /** Sets the item from the list into a specific holder in the recyclerView
     * @param viewHolder
     * @param i the position within the list
     */
    @Override
    public void onBindViewHolder(@NonNull ViewHolder viewHolder, int i) {...}
    /** Shows the number of items in the list
     * @return the number of values in the list
     */
    @Override
    public int getItemCount() { return songCollection.size(); }
    /** Interface to set the onClick method in other classes*/

    public interface OnNoteListener{
        void onNoteClick(int position);
    }
}
```

Figure 3.13: SongAdapter class

As can be seen in Fig. 3.14 the mentioned interface, implemented in the adapter's class, allows to be overridden it in the *CollectionDysplayer* to set the methods called when an item is clicked in the `RecyclerView`. This will result in a much cleaner and organized code. In fact, the behaviour of a click is set in the class where it logically belongs instead of in the RecyclerView adapter.

```java
/** Activity that displays music collection and allows to start a song in it*/
public class CollectionDisplayer extends AppCompatActivity implements SongAdapter.OnNoteListener {
    protected RunBPMapp model;
    private ImageButton buttonRun;
    private RecyclerView recyclerView;
    private RecyclerView.Adapter adapter;
    private RecyclerView.LayoutManager layoutManager;
    private static final String TAG = "MainActivity";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_collection_displayer);
        hideSystemUI();
        model = (RunBPMapp) getApplication();
        recyclerView = findViewById(R.id.recyclerView);
        layoutManager = new LinearLayoutManager( context: this);
        adapter = new SongAdapter(model.getSongManager().getSongCollection(),  onNoteListener: this);
        recyclerView.setLayoutManager(layoutManager);
        recyclerView.setAdapter(adapter);
        buttonRun = findViewById(R.id.buttonRun);
        model.setAnimation((AnimationDrawable) buttonRun.getBackground());
        buttonRun.setOnClickListener((v) -> { finish(); });
    }

    /**Hides navigation bar*/
    private void hideSystemUI() {
        getWindow().getDecorView().setSystemUiVisibility(View.SYSTEM_UI_FLAG_LAYOUT_STABLE
                | View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
                | View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN
                | View.SYSTEM_UI_FLAG_HIDE_NAVIGATION // hide nav bar
                | View.SYSTEM_UI_FLAG_FULLSCREEN // hide status bar
                | View.SYSTEM_UI_FLAG_IMMERSIVE);
    }

    /** Stets the behaviour of the click on one of the RecyclerView's rows starting the related song
     * @param i the position in the ArrayList
     * */
    @Override
    public void onNoteClick(int i) {
        model.resetMediaPlsyer();
        model.playMusic(model.getSongManager().getSong(i));
        Log.d(TAG,  msg: "onNoteClick1: " + model.getSongManager().getSong(i));
        Log.d(TAG,  msg: "onNoteClick2: " + model.getPlayedSongName());
    }
}
```

Figure 3.14: CollectionDysplayer activity class

Another aspect interest is how it has been achieved the matching between the songs and the user pace.

Firstly as it will be in dept explained in the next section, it has implemented a `SensorEventListener` to receive the detected steps, then each time a step is detected it is increased a counter instantiated in the *MainActivity* class, where the listener is implemented. It has not been implemented in the *RunBPMapp* application class since

it has to be instantiated in an activity.

```
/** Displays the song name when a song is played*/
public void setPlayedSong(){
    if(model.isPlaying()){
        playedSongText.setText(model.getPlayedSongName());
    }
}
/** Sets the right img. for the play/stop button*/
public void setPlayButtonImg() {
    if (model.isPlaying()) {
        buttonPlay.setImageResource(R.drawable.stop_button);
    } else {
        buttonPlay.setImageResource(R.drawable.play_button);
    }
}
/** Sets the right img. for the voiceOn/Off button*/
public void setVoiceButton() {
    if (voiceEnabled) {
        buttonVoiceOn.setImageResource(R.drawable.voice_off);
    } else {
        buttonVoiceOn.setImageResource(R.drawable.voice_on);
    }
}

/** Requests user permission to  access external storage and if allowed sets the songCollection*/
public void getExternalStoragePermission() {
    Dexter.withActivity(this).withPermission(Manifest.permission.READ_EXTERNAL_STORAGE).withListener(new PermissionListener() {
        @Override
        public void onPermissionGranted(PermissionGrantedResponse response) {
            //            songManager = new SongCollectionManager();
            model.getSongManager().setSongCollection();
        }
        @Override
        public void onPermissionDenied(PermissionDeniedResponse response) {
        }
        @Override
        public void onPermissionRationaleShouldBeShown(PermissionRequest permission, PermissionToken token) {
            token.continuePermissionRequest();
        }
    }).check();
}

/** Counts the number of steps in 30 sec. and calculates the steps per minute*/
public void calculateStepsPerMin() { stepCountHandler.postDelayed(stepCountRunnable, delayMillis: 30000); }

/** Runnable object that after and every 30 sec resets step count, plus sets, displays and if allowed pronounces the step per minute*/
private Runnable stepCountRunnable = () -> {
        model.setSpm(stepCount*2);
        Log.d(TAG, msg: "spm: " + String.valueOf(model.getSpm()));
        spmText.setText("Steps Per Minute: " + model.getSpm());
        if (stepCounterTTScorrectlySetted && voiceEnabled){
            pronounceSPM();
        }
        Toast.makeText( context: MainActivity.this, text: "spm: " + model.getSpm(), Toast.LENGTH_SHORT).show();
        stepCount = 0;
        stepCountHandler.postDelayed( r: this, delayMillis: 30000);
};

/** Initialises the TextToSpeech if English language available*/
public void setStepCounterTTS(){
    stepCounterTTS = new TextToSpeech( context: this, (status) -> {
            if (status == TextToSpeech.SUCCESS) {
                int result = stepCounterTTS.setLanguage(Locale.ENGLISH);

                if (result == TextToSpeech.LANG_MISSING_DATA
                        || result == TextToSpeech.LANG_NOT_SUPPORTED) {
                    Log.e( tag: "TTS",  msg: "Language not supported");
                } else {
                    stepCounterTTScorrectlySetted = true;
                    voiceEnabled = true;
                    setVoiceButton();
                    buttonVoiceOn.setEnabled(true);
                }
            } else {
                Log.e( tag: "TTS",  msg: "Initialization failed");
            }
    });
}
```

Figure 3.15: Second part of MainActivity class

The value of the mentioned counter is then every 30 seconds passed to the *RunBPMapp* class as official step count and then reset to 0, to count the steps of the following 30 seconds again.

This is achieved with the use of a `Handler` to manage a thread, to which runnable objects, with the instruction above, are passed. It is possible to observe the used methods carefully in Fig. 3.15 It has to be furthermore mentioned how the `READ_EXTERNAL_STORAGE` permission has been asked and therefore granted or denied. Here has been very helpful the *Dexter* library available on *GitHub* that provide methods to achieve that. It can indeed be seen again in Fig. 3.15 how with the

method `getExternalStoragePermission` it is possible to manage the whole process in a simple and clean manner.

```java
/** Class that represent the song's collection in the device.*/
public class SongCollectionManager {
    private ArrayList<Song> songCollection;

    /** Class constructor*/
    public SongCollectionManager() { songCollection = new ArrayList<>(); }

    /** For all the songs in the device creates Song object and add to the collection*/
    public void setSongCollection() {
        ArrayList<File> audioFiles = readOnlyAudioFiles(Environment.getExternalStorageDirectory());
        Log.d(TAG,  msg: "setSongCollection: audioFiles " + audioFiles);
        Log.d(TAG,  msg: "setSongCollection:  songCollection" + songCollection);
        for (File f : audioFiles) {
            Song song = new Song(f);
            Log.d(TAG,  msg: "setSongCollection: song " + song);
            songCollection.add(song);
            Log.d(TAG,  msg: "setSongCollection:  songCollection" + songCollection.size());
        }
        Log.d(TAG,  msg: "setSongCollection:  END!!!!");
    }
    /** Scans all the audios in the device, except for hidden folders and files and returns song's collection
     * @param file file in the device
     * @return ArrayList
     */
    public ArrayList readOnlyAudioFiles(File file) {
        ArrayList<File> audioList = new ArrayList<>();
        File[] allFiles = file.listFiles();
        for (File f : allFiles) {
            if (f.isDirectory() && !f.isHidden()) {
                audioList.addAll(readOnlyAudioFiles(f));
                 Log.d(TAG,  msg: "readOnlyAudioFiles: " + f);
            } else {
                if (!f.isHidden()){
                    if (f.getName().endsWith(".mp3") || f.getName().endsWith(".wav") || f.getName().endsWith(".wma")){
                        audioList.add(f);
                    }
                }
            }
        }
        return audioList;
    }
    /** Returns the song with the closest BPM to SPM, the first one will be the slowest
     * @param spm steps per minute
     * @return Song
     */
    public Song getClosestSongToSpm(int spm){
        Log.d(TAG,  msg: "getClosestSongToSpm: 0 step");

        int distance = Math.abs(songCollection.get(0).getBPM() - spm);
        Log.d(TAG,  msg: "getClosestSongToSpm: 1 step");
        int idx = 0;
        for(int c = 1; c < songCollection.size(); c++){
            int cdistance = Math.abs(songCollection.get(c).getBPM() - spm);
            if(cdistance < distance){
                idx = c;
                distance = cdistance;
            }
        }
        Log.d(TAG,  msg: "getClosestSongToSpm: 2 step");
        Song song = songCollection.get(idx);
        return song;
    }
    /** Returns song's collection
     * @return ArrayList
     */
    public ArrayList<Song> getSongCollection() { return songCollection; }

    /** Returns song at specific position
     * @param i position
     * @return Song
     */
    public Song getSong(int i){
        return songCollection.get(i);
    }
}
```

Figure 3.16: Complete SongCollectionManager class

The process of selecting the most appropriate song, as well as reading and collecting in a list all the available audio files it is instead handled by the `SongCollectionManager` class.

It is indeed collected all the "mp3", "wav" and "wma" files present in any folder of the smartphone except for hidden folders or file.

As can be observed in Fig. 3.16 the song with the closed BPM is then found calculating the absolute value of the difference between a song and the actual number of steps per minute; this process is repeated for all the available songs and then returned the closest one.

In the Fig. 3.17 it is then implemented a method to play a song, this will actually call itself again at the completion of the played song, but passing as its parameter, the method to obtain the closest song.
In doing so, this method is more reusable as it can be used to select a song from the displayed list as well as when the start button is pressed(and hence requested to play the closest song) and no matter what is the first usage then it will always continue playing the more appropriate song one after the other.
This will continue until the stop button is pressed or the user selects a new song.

It is visible how being this a prototype at the moment it does not have a much-complicated structure, but it has been developed in a way that it would be possible to easily implement new functionalities without having to modify the overall design.
Finally, to clarify any further doubts or curiosities and conclude this overview, it will be provided the full copy of the remaining code in the Appendix section.

```java
/** Application class that hold the Media Player and the songManager*/
public class RunBPMapp extends android.app.Application {
    private SongCollectionManager songManager;
    private MediaPlayer mediaPlayer;
    private boolean isPlaying;
    private int spm;
    private String playedSong;
    /** Application class constructor*/
    @Override
    public void onCreate() {
        super.onCreate();
        songManager = new SongCollectionManager();
        mediaPlayer = new MediaPlayer();
        isPlaying = false;
        spm = 0;
        playedSong = "";
    }
    /** Returns the songCollectionManager
     * @return the SongCollectionManager
     */
    public SongCollectionManager getSongManager() { return songManager; }
    /** Returns the MediaPlayer
     * @return the MediaPlayer
     */
    public MediaPlayer getMediaPlayer() { return mediaPlayer; }
    /** Returns if the MediaPlayer is playing
     * @return boolean
     */
    public boolean isPlaying() { return isPlaying; }
    /** Stops the MediaPlayer*/
    public void stopMusic(){
        mediaPlayer.stop();
        isPlaying=false;
    }
    /** Resets the MediaPlayer*/
    public void resetMediaPlayer() { mediaPlayer.reset(); }
    /** Returns the Steps per min.
     * @return int
     */
    public int getSpm() { return spm; }
    /** Returns the song that is playing
     * @return String
     */
    public String getPlayedSongName() { return playedSong; }
    /** Sets the steps per min.
     * @param spm
     */
    public void setSpm(int spm) { this.spm = spm; }
    /** Starts selected song reproduction followed by the most appropriate ones
     * @param song Song object
     */
    public void playMusic(Song song){
        try {
            mediaPlayer.setOnCompletionListener((mPlayer) -> {
                    isPlaying = false;
                    mediaPlayer.reset();
                    playMusic(songManager.getClosestSongToSpm(spm));
            });
            Toast.makeText(getApplicationContext(), song.getName(), Toast.LENGTH_SHORT).show();
            mediaPlayer.setDataSource(song.getFile().toString());
            mediaPlayer.prepare();
        } catch (IllegalArgumentException e) {
            e.printStackTrace();
        } catch (Exception e) {
            System.out.println("Exception of type : " + e.toString());
            e.printStackTrace();
        }
        mediaPlayer.start();
        playedSong = song.getName();
        isPlaying = true;
        Toast.makeText(getApplicationContext(), text: "Start new song: " + playedSong, Toast.LENGTH_SHORT).show();
    }

    /** Sets animation of different objects
     * @param a AnimationDrawable
     */
    public void setAnimation(AnimationDrawable a) {
        AnimationDrawable animation = a;
        animation.setExitFadeDuration(2000);
        animation.setEnterFadeDuration(4000);
        animation.start();
    }
}
```

Figure 3.17: Complete RunBPMapp class

Chapter 4

Research Methods

In this chapter will be analysed the process and the decision carried out in order to obtain the results subsequently explained.
As mentioned in the literature review, the best possible achievable goal would be a perfect matching between beats and steps, to be able to then through experiments and testing find the most suitable solution for the user. Although this was difficultly achievable over the short time available, however, the perspective has always been to produce the maximum achievable amount of work in this direction, instead to aim of a more comfortable and accessible target, but not being the optimal solution. In this way, this preliminary work can be useful as a foundation to further improvements.
This prespective is mainly reflected in the choice of detecting single bets instead of using more simple classification like genre and in the methodologies used to detect each step instead of just aiming for a count of them. Now, since these are the two main areas of investigation, for cohesion purposes will be treated one after another to avoid confusion between the two.

4.1 Steps Detection and Counting

Since the expressed objectives require to experiments the full process of research including dataset development, it was necessary to design an experiment to acquire enough quality data to train a ML model over it.

Following all this process, as well as the methodologies attempted and implemented, will be detailed in this section.

4.1.1 Experimental Set-Up

As mentioned in the requirements section, the assumption of the smartphone held in the pocket has been made. This finds its reason manly in the simplification of having data that are as uniform as possible. Considering indeed as an example the placing the smartphone on an armband and focusing over one side of the body, it is immediately noticeable how movements of arm and thigh during the running activity are in opposite directions. This will result in opposite values of accelerometer and gyroscope sensors.
There could still possibly be found a correlation between the opposite arm and tight. However, seems more appropriate, in order to achieve the maximal precision, to first apply a model to detect the kind of placement, and only then apply a step detection with weights and parameters studied specifically for that placement.
Therefore for the time being the focus will be on just on what resulted the most popular position used and bearing that in mind it is now possible to outline the characteristics of this experiment.
Due to the high level of attention and concentration required to detect each step, it has been decided to involve in the recordings of data just the researcher itself, to have the highest possible confidence in the data accuracy.

It has been then established to consider running sessions limited to a max of ten minutes, resulted after some experiments as the time beyond which the accuracy tent to decrease. It is indeed not that easy to keep track of all the steps, especially when the speed increase and this has been the reason why has been chosen to record only the steps performed by the right leg.
Moreover, to increase the level of abstraction, the experiments had been repeated under different condition. It have been recorded multiple running sessions over various terrains (concrete, grass, beaten earth and treadmill), with diverse speeds (from a

minimum of 5 km/h to a max of 15km/h) and with different level of incline, workloads and rhythms, taking care to always include several variations of pace.
The reason behind this choice is to simulate the majority of possible use scenarios, considering peculiar people habits. Moreover, notable relevance has been given to record data holding the device in a constrained and especially unconstrained manners, reflected by the use of diverse type of clothes(from gym shorts to jeans). This is particularly significant because the data recorded from sensors in these two diverse ways will be quite different and being able to develop a model that produce results regardless that would be an exciting result.

It will be lastly specified that some attention has been reserved to have a balanced mix of all the previously mentioned condition to avoid that one of them could particularly affect the prediction algorithm.

4.1.2 Data Acquisition

With in mind the outline of the experiment it is now possible to focus on which data and how has been recorded.
The first thing to cover is the device used for all the carried out tests and recordings, i.e. a *"Redmi Note 7"* by *Xiaomi* with these specs available in Tab. 4.1.

Processor	**RAM**	**OS**	**Accelerometer**	**Gyroscope**
Snapdragon 660 AIE (2.2GHz)	4GB	Android 9.0	Yes	Yes

Table 4.1: Test Device specs

Android devices allow access to a variety of different type of data gathered from sensors. Even though it is possible that some devices do not provide with them all, however, nowadays most of them are available.
What are relevant for this project are the provided *motion sensors* and this a brief overview of them all:

- `TYPE_ACCELEROMETER`, provide three values of acceleration force along the 3 axes

(including gravity);

- `TYPE_ACCELEROMETER_UNCALIBRATED`, provide six values of acceleration along the 3 axes with and without estimated bias compensation;
- `TYPE_GRAVITY`, provide three values of force of gravity along the three axes;
- `TYPE_GYROSCOPE`, provide three values of rate of rotation around the 3 axes;
- `TYPE_GYROSCOPE_UNCALIBRATED`, provide six values of rate of rotation with and without drift compensation around the 3 axes;
- `TYPE_LINEAR_ACCELERATION`, provide three values of acceleration force along the 3 axes (excluding gravity);
- `TYPE_ROTATION_VECTOR`, provide three values of rotation vector component along the 3 axes ($axis * sin(\theta/2)$);
- `TYPE_STEP_COUNTER`, return number of steps taken by the user since the last reboot while the sensor was activated;
- `TYPE_STEP_DETECTOR`, return the 1 value each time a step is taken by the user;

In Fig. 4.1, it is outlined how the mentioned axes are oriented, and it must be specified that they will not swap when the device's screen orientation changes.

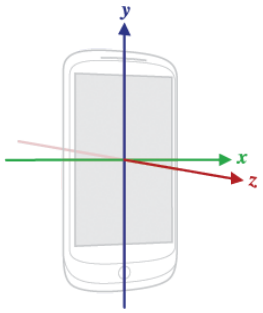


Figure 4.1: Android sensors axes orientation

Android has a specific class to hold sensors values called `SensorEvent` that have the following fields, a multi-dimensional array of values(except for the last two that has just a single value), the accuracy, the sensor type and a timestamp (time in nanoseconds at

which the event happened).
The way to access them is via the provided abstract class `SensorManager` with the method `getDefaultSensor()`, specifying the desired type choosing from the list previously provided. Then, once acquired, it must be initialized a `SensorEventListener`, interface necessary to receive notifications from the SensorManager when there is new sensor data available. Finally, the listener must be registered in the `SensorManager` with the method `registerListener()` where it can also be specified the type of desired sampling frequency between the provided:

- `SENSOR_DELAY_NORMAL` suitable for monitoring typical screen orientation changes and uses a delay of 200,000 microseconds;
- `SENSOR_DELAY_UI` uses a delay of 60,000 microsecond delay;
- `SENSOR_DELAY_GAME` uses a delay of 20,000 microsecond delay;
- `SENSOR_DELAY_FASTEST` uses a delay of 0 microsecond delay;

It must be although clarified that these are just the suggested delay as it can be usually lower, as the specified value can be more appropriately considerate as the max acceptable delay.
Each `SensorEvent`s can be finally accessed with the `onSensorChanged()` method provided by the `SensorEventListener` interface that will be called each time there is a new event, this method can be hence overridden to specify the desired behaviour, however it must be cleared that "on changed" is somewhat of a misnomer, as this will also be called if we have a new reading from a sensor with the exact same values (but a newer timestamp) [6]. In this project have been used `TYPE_ACCELEROMETER` and `TYPE_GYROSCOPE`, selected as the right trade-off, between rawness and amount of preprocessing already applied; furthermore, the number has been restricted to two for this main reasons:

1. to limit the number of features, maintaining at a reasonable level the amount of computational power and time required to run the NN, considering that had

been tested over a laptop GPU and a fortiori since it should ideally run on a smartphone GPU or worst case scenario CPU;

2. to non-overload the amount of memory required by the app, since the process of record and write data is a draining operation to manage and Android can arbitrarily shut down a process and kill the app components running it, whenever memory is low or required by other processes.

With this brief overview, it is now clear how it is possible to records sensors data. However, it has to be explained the methodology used to record the actual values of these when a step is manually recorded pressing of the media button available in most of the earphones.
Firstly, ass shown in Tab. 4.2 it has been decided to build a database with these headers (*Accelerometer along axes X, Y and Z, Gyroscope rate of rotation around axes X, Y and Z, System Time, instance of Date precise to milliseconds, manually detected steps, Android detected steps and type of event.*) for each data entry, whatever is the triggering event, no matter if Android's Accelerometer, Gyroscope or step detector event or a manually detected step.

aX	aY	aZ	gX	gY	gZ	systemTime	Date	manualStep	androidStep	eventType
6.366.455	16.283.463	31.590.118	0.5065918	27.833.252	-4.200.012	1,56218E12	2019-07-03 19:37:48.734	0	0	accelerometer
6.366.455	16.283.463	31.590.118	0.6930237	27.993.011	-4.180.847	1,56218E13	2019-07-03 19:37:48.734	0	0	gyroscope
6.366.455	16.283.463	31.590.118	0.6930237	27.993.011	-4.180.847	1,56218E14	2019-07-03 19:37:48.759	1	0	manualStepDetector
6.366.455	16.283.463	31.590.118	0.8517456	27.662.811	-41.542.206	1,56218E15	2019-07-03 19:37:48.774	0	0	gyroscope
6.366.455	16.283.463	31.590.118	0.98384094	27.215.424	-4.131.836	1,56218E16	2019-07-03 19:37:48.775	0	0	gyroscope

Table 4.2: Dataset head

Then the manually detected steps are identified through the `KeyEvent.Callback` interface, so that when the `onKeyDown()` method is called a new data-point is created with the most recent values.
In order to achieve that it was crucial to be capable of have all the sensors' values related to each data entry, this has been achieved implementing a class to temporarily store and update those values each time new ones are available. It is then understandable why it is needed the lowest delay possible, therefore `SENSOR_DELAY_FASTEST`, as any delay can compromise the accuracy of data. Indeed if when a step is recorded the

sensors have new values, but due to delay, are not yet notified, it will results in the wrong matching between steps and peeks of acceleration or rotation.
It has been then decided that the best available methodology to store and collect these data was on a *CSV files* that will be programmatically sent to the researcher email once a recording session is declared ended by pressing a button on the screen. This choice can be explained by the purpose of potentially, in future experiments, scale the recording process to multiple subjects, increasing the level of abstraction.
Different approaches have been tested to find the optimal writing methodology. It has indeed been noticed how, short delay time combined with multiple sensors is reflected in many entries per millisecond, as visible in the first two row of Tab. 4.2, causing the app to be shut down forcibly due to long response time Fig. 4.2.

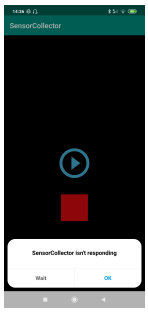


Figure 4.2: SensorCollector isn't responding

This behaviour is registered implementing the traditional and often suggested methodology [15] to access multiple sensor, i.e by using a single `SensorManager` and `SensorEventListener` and then distinguish events by their type in the `onSensorChanged()` method as even recommended in Listing 4.1 [34].

```
private SensorManager manager;
private SensorEventListener listener;

manager = (SensorManager) this.getSystemService(Context.SENSOR_SERVICE);
listener = new SensorEventListener() {
        @Override
        public void onAccuracyChanged(Sensor arg0, int arg1) {
        }
        @Override
        public void onSensorChanged(SensorEvent event) {
```

```
                Sensor sensor = event.sensor;
                if (sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
                }
                else if (sensor.getType() == Sensor.TYPE_GYROSCOPE) {
                }
        }
}
manager.registerListener(listener,
        manager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER),
        SensorManager.SENSOR_DELAY_GAME);
manager.registerListener(listener,
        manager.getDefaultSensor(TYPE_GYROSCOPE),
        SensorManager.SENSOR_DELAY_GAME);
```

Listing 4.1: Recommended imlementation to record multiple sensors

Furthermore, another anomaly has been noticed, i.e. the android detected steps seems to be grouped in the recording or writing phase, resulting in blocks of steps hypothetically performed with unreasonable timing.

aX	aY	aZ	gX	gY	gZ	systemTime	Date	manualStep	androidStep	eventType
0.66770935	-0.7369232	12.436.523	0.0126953125	-0.08067322	-11.791.077	1,56347E12	2019-07-18 19:17:09.633	0	0	accelerometer
-0.66770935	-0.7369232	12.436.523	0.08087158	-0.26176453	-11.759.186	1,56347E12	2019-07-18 19:17:09.634	0	0	gyroscope
-0.45222473	-0.8422699	10.257.721	0.14904785	-0.44180298	-11.801.758	1,56347E13	2019-07-18 19:17:09.638	0	0	gyroscope
-0.45222473	-0.8422699	10.257.721	0.08087158	-0.26176453	-11.759.186	1,56347E14	2019-07-18 19:17:09.638	0	0	accelerometer
-0.45222473	-0.8422699	10.257.721	0.14904785	-0.44180298	-11.801.758	1,56347E15	2019-07-18 19:17:09.642	0	1	androidStepDetector
-0.45222473	-0.8422699	10.257.721	0.14904785	-0.44180298	-11.801.758	1,56347E16	2019-07-18 19:17:09.642	0	1	androidStepDetector
-0.45222473	-0.8422699	10.257.721	0.14904785	-0.44180298	-11.801.758	1,56347E17	2019-07-18 19:17:09.643	0	1	androidStepDetector
-0.45222473	-0.8422699	10.257.721	0.14904785	-0.44180298	-11.801.758	1,56347E18	2019-07-18 19:17:09.643	0	1	androidStepDetector
-0.45222473	-0.8422699	10.257.721	0.14904785	-0.44180298	-11.801.758	1,56347E19	2019-07-18 19:17:09.643	0	1	androidStepDetector
-0.45222473	-0.8422699	10.257.721	0.14904785	-0.44180298	-11.801.758	1,56347E20	2019-07-18 19:17:09.643	0	1	androidStepDetector
-0.45222473	-0.8422699	10.257.721	0.14904785	-0.44180298	-11.801.758	1,56347E21	2019-07-18 19:17:09.643	0	1	androidStepDetector
-0.45222473	-0.8422699	10.257.721	0.14904785	-0.44180298	-11.801.758	1,56347E22	2019-07-18 19:17:09.643	0	1	androidStepDetector
-0.45222473	-0.8422699	10.257.721	0.14904785	-0.44180298	-11.801.758	1,56347E23	2019-07-18 19:17:09.645	0	1	androidStepDetector
-0.45222473	-0.8422699	10.257.721	0.14904785	-0.44180298	-11.801.758	1,56347E24	2019-07-18 19:17:09.645	0	1	androidStepDetector
-0.26786804	-0.95718384	0.80311584	0.14904785	-0.44180298	-11.801.758	1,56347E25	2019-07-18 19:17:09.646	0	0	accelerometer
-0.10507202	-10.433.807	0.5589142	0.14904785	-0.44180298	-11.801.758	1,56347E26	2019-07-18 19:17:09.646	0	0	accelerometer
-0.10507202	-10.433.807	0.5589142	0.20443726	-0.61650085	-11.833.801	1,56347E27	2019-07-18 19:17:09.646	0	0	gyroscope
0.019424438	-1.103.241	0.33146667	0.25024414	-0.7826843	-1.184.433	1,56347E28	2019-07-18 19:17:09.649	0	0	accelerometer

Table 4.3: Android detected steps

It can be seen in Tab. 4.3 how ten steps are recorded over three milliseconds, and this is a behaviour that is repeated several times over any recording session. It is hard to precisely understand the reasons of these behaviours since the programmer does not have the full control when hardware-based composed are involved; however, some investigation has been carried on, and three possible superficial causes have been identified:

1. an overlapping of different output stream trying to write at the same time on one CSV file;

2. too numerous operations addressed to the *UI Thread*(main thread of execution for the application, where most of the application components as Activities, Services, ContentProviders and BroadcastReceivers are created and run, alongside any system calls to those) to be efficiently handled;

3. too many events in a short space of time,to be handled by a single `SensorManager` or `SensorEventListener`.

The first attempt tested to exanimate the first option has been writing each sensor on a distinct file. However, no considerable improvements have been noticed; indeed if more than one sensor use the `SENSOR_DELAY_FASTEST` the application will not respond, an indication that this is not the triggering cause.
Different approaches have therefore been experienced to move the workloads away from the UI Thread, in line with the Official Android documentation:

"In any situation in which your app performs a potentially lengthy operation, you should not perform the work on the UI thread, but instead create a worker thread and do most of the work there." [5]

As a results of some experiments over `AsyncTask` and `HandlerThread` classes the best option available seemed the `ThreadPool`. The reason behind this choice is the nature of the work that needs to be offloaded. It can indeed be observed a high number of small distinct task, i.e. having to write a new entry on the CSV file. This is is particularly well-suited to the `ThreadPoolExecutor` class, that indeed *"address two different problems: they usually provide improved performance when executing large numbers of asynchronous tasks, due to reduced per-task invocation overhead, and they provide a means of bounding and managing the resources, including threads, consumed when executing a collection of tasks."* [7]
This is achieved by collecting the required tasks in a queue(in the present case

`LinkedBlockingQueue`). New threads are created with the following logic, every time a new task is delegated to the thread pool a new one is created even if idle threads exist, this happens until the `corePoolSize`(minimum number of threads to keep in the pool) is reached, then new threads are created only if the queue of tasks is full, until the `maximumPoolSize`(maximum number of threads allowed in the pool) is reached.
Some test to monitor its functioning has been made, and how shown in Tab. 4.4. It has been noticed that both the number of threads alive and working increase significantly. Indication of the fact that probably was too much to handle only by the UI Thread.

Without ThreadPoolExecutor		**With ThreadPoolExecutor**	
Max number of alive Threads	Range of working Threads	Max number of alive Thread	Range of working Threads
13-14	7-9	21-22	7-17

Table 4.4: Comparison overview of Sytem threads

Finally to address the last possible cause has been implemented distinct `SensorManager` and `SensorEventListener` for each of the motion sensor recorded, and that seemed to be what more than anything else make the most visible improvements, resulting in the prime candidate of causing the undesired behaviour.
That being said, it still seems recommended to use a ThreadPool to have a more canonic implementation, avoiding to overload the UI Thread and therefore creating possible delays in the recordings of data.
Even though, with this configuration, being able to record and write all the desired sensors without drawbacks of any kind, still it has not been found a definitive explanation to the weird exposed behaviour of multiple steps detected by Android in incredibly small amount of time.
As far as has been possible to exanimate, two potential theories the could be peddled:

- it is due to the decision of pairing it with the SystemTime and an instance of Date when the `onSensorChanged()` method is called, instead of using its available timestamp. This has been done to have a uniform temporal reference common to all the datapoint, whatever the triggering event. It could be therefore possible that the anomaly is addressed to the SystemTyme acquisition, that could oddly happen with some delay. However, it is quite unlikely since it would be

anyhow strange that many sequential steps happen without being interspersed by accelerometer or gyroscope events.

- This leads to the second and more plausible theory, i.e. it is the regular Android step counter behaviour, it is indeed possible that it is not that precise. Indeed, more than accurately detect precisely when each step is performed, mostly count and return them together sequentially. This supposition is also supported by the fact that the same behaviour is also seen in the debugger and running windows logging and printing the event type every time a new one is returned.

This is another motivation that leads to the necessity of using manually detected steps as labels for the training process of the ML algorithm since it is impossible to rely entirely on Android's one.

It has then been recorded several running session where all the steps have been manually detected, specifying how if a misdetection has been noticed during the experiment performance, the corresponding session has been trashed.
This results of this process are 22 CSV files, representing each of the best sessions.

4.1.3 Data Preparation

All the data file received by email had been saved on a specific folder, naming each file with the date of when has been recorded. Then a script has been made in order to collect all these on a single and more manageable dataset file.
Moreover has been chosen to take just the portion of these files from the first to last step. It has been avoided to transfer in the final dataset not useful and noisy data from the start and end of the session. Doing that, movements that can create confusion in the training process as placing the phone in the pocket and take it out to stop the recording, are not taken into account.
This, however, considerate the data as a time series, could possibly lead to confusion as in the transition between a session and the following one, two consecutive data-point will record a step and could be misleading. However this, as all the preprocessing methodologies, will be treated in the next section.

4.1.4 Data Analysis and Preprocessing

The resulting final dataset it is then composite of 3 098 922 from which has been dropped the last 22 entries in order to have a more manageable and most importantly easily divisible number.

Total entries	Android's detected steps	Manually detected steps	Accelerometer events	Gyroscope events
3098900	16636	8424	1536917	1536923

Table 4.5: Final dataset description

Analysing the Tab. 4.5 it is possible to notice how the Android's detected steps are vaguely double the number of the manual ones. This could mean that potentially are detected the steps of both legs or that the step movement is split in two, i.e. the rising and descending motion of the leg. However, considering its uneven behaviour, it is hard to believe it is actually one of these options, inclining more for a high number of false positive.

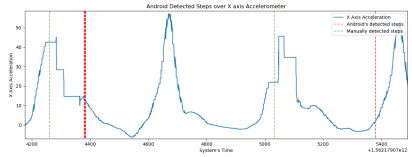


Figure 4.3: Acceleration over the X axis

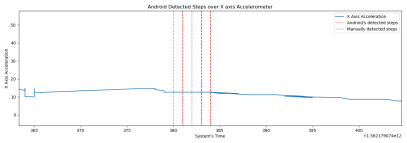


Figure 4.4: Acceleration over the X axis

To support this theory it can bee sees in Fig. 4.3 how looking closely at a step level not all the vertical lines, indicating an Android detected step, have the same thickness, this is because, how shown in Fig. 4.4, these are actually five steps recorded in less than a second.

It can also be noticed how these are not detecting steps from both legs, as the four observable peeks of acceleration visible in Fig. 4.3 corresponding respectively to (right, left, right, left steps) are not anyhow matched by the Android's detected steps. This is furthermore suggesting how what at first could have seemed an issue in the methodologies used in the data recording it is in all likelihood the typical behaviour of Android.

Moreover, it must be considerate that unlike the plots presented until now, the ML model will be fitted with just the sequence of data-points, without considering the time spacing between them. However, thanks to the low delay obtained in the recording phase, observing a small section of 5000 samples, the difference between Fig. 4.5 and Fig. 4.6 it is hardly noticeable.

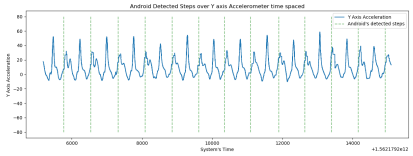


Figure 4.5: Acceleration over the Y axis time spaced

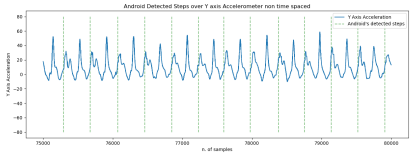


Figure 4.6: Acceleration over the Y axis non time spaced

Before moving on to the proper data analysis, it needs to be explained in more detail how the manually detected steps have been used as labels. Having each data point in the *ManualStep* column a 1 if it is a step and a 0 it is a different event, the simplest way to use these is to consider them as two classes: *"Step"* and *"Non Step"*. It is however easily recognizable, bearing in mind Tab. 4.5, how there is a significant unbalance between the two classes, having 8 424 *Step* against 3 090 476 *Non Step*. It is also evident in Tab. 4.6 how the accuracy of matchings between acceleration peeks and detected steps is good, but obviously not perfect. This can be addressed to the human imprecision and still some delay between the moment when the earphone button is pressed and when it is recorded.

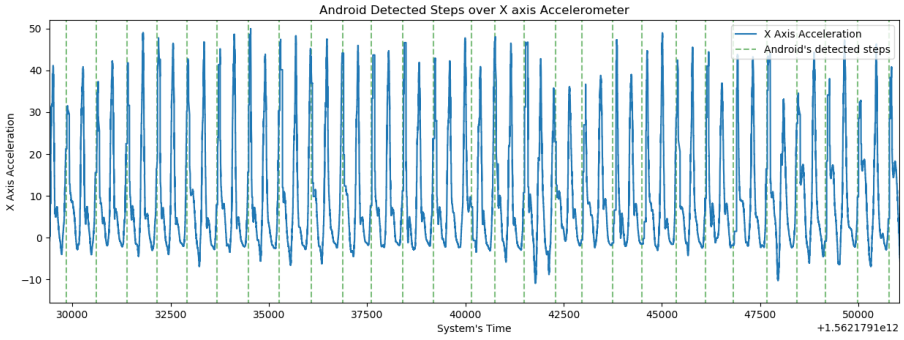


Figure 4.7: Accuracy of manual detection on axis X acceleration

A possible solution to the accuracy problem could be reducing the jitters of the line

chart applying statistical calculation as the moving average. *Pandas* Python's library allows this operation combining two of its core functionality: `rolling()` and `mean()`, the first one allows to perform calculations over a rolling window of desired size and the second one return the mean value in the window.
This, however, do not anyhow reduce the unbalancing problem, therefore has been considerate to develop a revisited implementation of it to affect both the enumerated issues.

```
def mov_avg_on_features(df, mean_value):
        df_avg = df.groupby(np.arange(len(df.index))//mean_value).mean()
        return df_avg
```

Listing 4.2: Reinterpretation of Moving Average over Features

As can be seen in Listing 4.2 the mean value of all the features(Accelerometer and Gyroscope values) it is not calculated over a rolling window. Instead, the values are grouped according to a selected `mean_value` and then over these, it is calculated the average.
The main reason to prefer this approach is how labels can be treated:

```
def mov_avg_on_labels(labels, mean_value):
        lab = lab.groupby(np.arange(len(labels.index))//mean_value).max()
        return lab
```

Listing 4.3: Reinterpretation of Moving Average over Labels

Indeed based on whether a group include a step or not, it will be returned just a value of 1 or 0. This two methods together will reduce the unbalancing of the dataset, reducing the number of total samples without however decrease the number of instances of the *Step* class.
Two different `mean_value` has been tried, with the following results:

Original Dataset		**Dataset resulted with mean_value of 3**		**Dataset resulted with mean_value of 5**	
Tot. samples	Steps	Tot. samples	Steps	Tot. samples	Steps
3098900	8424	1032950	8407	619750	8405

Table 4.6: Comparison of the effects of different mean_vaues over the Dataset unbalance

Nonetheless, it can be noticed a minor reduction, this is, however, a desirable consequence, since, as previously mentioned, the running session has been trimmed to the

first and last steps to have dataset containing only running data, therefore on the junctions two consecutive data-point will results as two steps one after the other. These are merged together resulting in a single entry.
This, however, comes along with the drawback of reducing the size of the dataset already not huge, making possibly the chose of a `mean_value` of 3 a more affordable option. Nevertheless, the unbalance, despite being reduced, it can still significantly influence the ML algorithm, producing biased results in the form of reaching a high accuracy without detecting any steps. Therefore has been decided to produce weights to each samples to rebalance the two classes of labels.
To achieve that it is particularly handy `compute_sample_weight()` method from the library *Scikit-learn* [39] that calculate the weight of each sample according to their classes, assigning a weight of **0.5041028048603133** to each *Non Step* and **61.43392411086** to each *Step*.

Finally considering Tab. 4.7 how Accelerometer and Gyroscope sensors can have a high variance in their measurements, to have more normalized data has been scaled to fit in a range between -1 and 1.

aX	**aY**	**aZ**	**gX**	**gY**	**gZ**
-0.34687805	15.105.514.999.999.900	-3.355.606	0.40753174	-1.105.957	-0.17225647
-0.34687805	15.105.514.999.999.900	-3.355.606	0.28501892	-10.462.951.999.999.900	-0.26280212
-0.34687805	15.105.514.999.999.900	-3.355.606	0.16464233	-10.143.433.000.000.000	-0.3789215

Table 4.7: First 3 rows of the Dataset averaged with a mean_value of 3

aX	**aY**	**aZ**	**gX**	**gY**	**gZ**
-0.004429966218252667	0.1927699504034695	-0.044657642542667686	0.01616198251301425	-0.03231623982860062	-0.009067390404706248
-0.004429966218252667	0.1927699504034695	-0.044657642542667686	-0.009908589590776085	-0.025434755076415377	-0.027389316035804814
-0.004429966218252667	0.1927699504034695	-0.044657642542667686	-0.021839117234798686	-0.019933896734393712	-0.03748778239513067

Table 4.8: First 3 rows of the scaled Dataset averaged with a mean_value of 3

Specifically, as can be noticed in Tab. 4.8, it has been used again the library *Scikit-learn* choosing the `MaxAbsScaler`, that "translates each feature individually such that the maximal absolute value of each feature in the training set will be 1. It does not shift/center the data, and thus does not destroy any sparsity." [2]. This will result in

a dataset scaled in a range between -1 and 1. This somehow, probably precisely for this property, generate better results in the training phase then other scalers like the `MinMaxScaler` that would produce a similar dataset selecting parameters of min -1 and max 1.

4.1.5 Data Classification and Training

Moving now on how the obtained data has been classified, the first thing to mention is that in this experiments it has been considerate only supervised learning algorithms, this is because the two problems lend themselves well to be labelled in a reasonably straightforward manner.
To approach the step detection problem, two main algorithms have been tested:

1. **Support Vector Machines**, particularly recommended for binary classification problems and proved to be able to get excellent results even using a small datasets.
2. **Long Short Term Memory Neural Network**, specially designed for time series problems, with the ability to remember previous temporal steps and produce prediction according to these.

The selection of these two is to provide an overview of two different possible approaches, considering data-point individually, therefore having a simple binary classification problem, or alternately thinking them as consecutive steps in time and hence dealing with a time series classification problem.

4.1.5.1 Support Vector Machines

Starting from SVM, the idea behind it can be simply explained as imagining to visualize each data-point in a n-dimensional space, where n is number of provided features, the classification is then performed searching the hyper-plane that divide the two classes as best as possible.
The name of the algorithm is indeed inspired by the *support vectors*, i.e. co-ordinates of each individual observation.

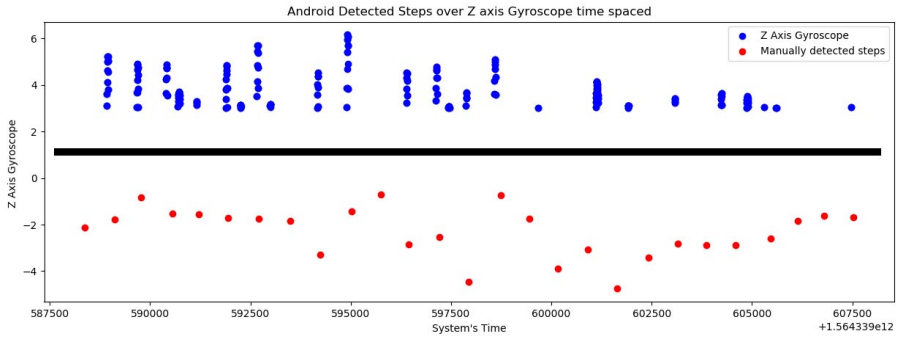


Figure 4.8: Hypothetical ideal Hyper-plane classification observation distribution

Simplifying this classification problem to just one feature to have just two dimensions in order to have a more straightforward visualisation. It can be then seen in Fig. 4.8 how ideally a hyperplane (identified by the black line) would separate two classes, as shown in the figure *steps* and *non steps* values from the Z-axis of the Gyroscope.

The equation of the hyperplane in a 2D space will hence be the equation of the line that divide two classes and is expressed as:

$$y = a * x + b$$

To find the line that in the best possible way defines the two classes, it is then just a matter of finding the optimal slope (a) and the intersection with the y axis (b).

This becomes an optimisation problem, i.e. find weights that allow the most significant possible margin between itself and any point within the training set.

Nevertheless, it can be imagined how, in reality, data will be very different from the one in Fig. 4.8. Indeed looking at the totality of data-point recorded by the gyroscope in Fig. 4.9 it is clear how a linear classification in 2D would not be possible, and this is the reason why it has been used six features. Therefore the hyperplane will be searched in a 6D space, hoping that considering more features there will be a more apparent distinction between *steps* and *non steps*.

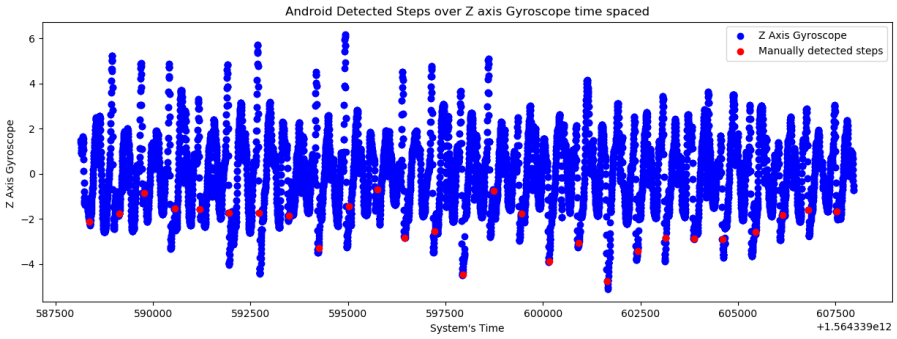


Figure 4.9: Real observation distribution of the rate of rotation over the Z axis

It has then been used *scikit-learn* library to build a SVM model to classify the dataset and considering the unbalancing between the labels to avoid to encounter just *non step* predictions some procedure was required.
Remembering now how it was previously mentioned, that labels were unbalanced, it will now be explained why that is so relevant in this context.
This issue requires the adding of weights to each sample to balance the algorithm bias. Otherwise, since predictions are merely based on mathematical optimisation, if a high accuracy can be achieved, always detecting *non steps*, the hyperplane parameters will be optimised in this direction.

Unfortunately, using *scikit-learn*, it has not been possible to assign weights neither at a class level (assigning the computed weights as parameter `class_weight=` initializing the SVM model), nor at samples level (assigning individual weights to each sample and fitting them as parameter `sample_weight=` alongside the training dataset and labels). It has been indeed observed how trying to run the model would result in an endless process, even letting it run for several hours.
An alternative solution to this issue has been selecting from the dataset just the row containing a *step* and then add the same amount of *non step* rows, in order to balance the two classes.
The size of this created dataset has been then also doubled and multiplied by ten, always selecting different random *non step* rows, to test the learning over a diverse

amount of training data.

4.1.5.2 Support Vector Machine Results

Two typologies of SVM has been tested, the *linear SVM* and *non linear SVM* over the values of accelerometer and gyroscope on the three recorded axes and although the not brilliant performances shown in Tab. 4.9 was expected, however, the negligible improvements obtained with the *non linear SVM* in Tab. 4.10 has been a bit disappointing.
As can be noticed the *accuracy*, *precision* and *recall*, along with the record of the number of *real* and *predicted steps* has been measured over the three datasets. Nevertheless, being these appositely made up to training purposes, the algorithms have been tested even over the originally chronologically recorded and averaged data. This will simulate the behaviour of the model over real conditions and will provide a term of comparison with the LSTM algorithm.

	Created Balanced Dataset			Prediction over Balanced Validation					Prediction over Original Dataset					Prediction over Same Validation as LSTM				
	Total samples	Training	Validation	Steps	Predicted	Accuracy	Precision	Recall	Steps	Predicted	Accuracy	Precision	Recall	Steps	Predicted	Accuracy	Precision	Recall
Training over 10X Dataset	168140	117698	50442	25585	22987	0.70	0.73	0.65	8407	253934	0.75	0.02	0.65	2539	74495	0.76	0.02	0.61
Training over 2X Dataset	33628	23539	10089	5063	4666	0.71	0.73	0.67	8407	259276	0.75	0.02	0.66	2539	75642	0.75	0.02	0.62
Training over 1X Dataset	16814	11769	5045	2516	2320	0.71	0.73	0.67	8407	261426	0.75	0.02	0.66	2539	76444	0.75	0.02	0.62

Table 4.9: Prediction results of Linear SVM comparison

	Created Balanced Dataset			Prediction over Balanced Validation					Prediction Same Validation Dataset				
	Total samples	Training	Validation	Steps	Predicted	Accuracy	Precision	Recall	Steps	Predicted	Accuracy	Precision	Recall
Training over 10X Dataset	168140	117698	50442	25468	26950	0.77	0.76	0.80	2539	80292	0.75	0.03	0.82
Training over 2X Dataset	33628	23539	10089	5166	5422	0.74	0.74	0.77	2539	85207	0.73	0.02	0.77
Training over 1X Dataset	16814	11769	5045	2562	2585	0.75	0.75	0.76	2539	78277	0.75	0.02	0.75

Table 4.10: Prediction results of Non Lineas SVM comparison

The first noteworthy thing to notice is how the algorithm is not able to transfer the decent performances obtained over the training dataset to the real data noticeably in the drastic drop of precision and the astronomical number of step detected.
Moreover as could have been predicted the *non linear* classification produced better results over the validation dataset; nevertheless these are not entirely reflected over the original data.
It can also be seen in Tab. 4.10 how although the increase of the dataset's size has made it possible better results in the training evaluation dataset; however, these are

again not significantly reflected over the original dataset.
This somehow indicates how this approach as it has been applied is not suitable to this problem, as no matter how will be increased the amount of data recorded, it will not produce better performances.
A final consideration will be made on how the small increase in performances due to the use of a *non linear SVM* it is not worth in this scenario, as it is not proportional to the more time required to process the data, being this one significantly higher.

These results are not totally unexpected, as, how due to resource, time and even skills limitation, a series of imprecisions and simplification has been tolerated, starting from the methodologies adopted in the recording methods, to android sensor managing limitation, to finally data preprocessing.
As a simple example, a different methodology in the recording of steeps, it is believed that, will already significantly improve the level of classification prediction. It indeed can be noticed in Fig. 4.9 how even after the preprocessing of data, the recorded steps do not perfectly match with the peeks of rotation. It can be imagined how this is particularly relevant with a classification algorithm so highly based on the accuracy of data. Indeed it only has them to learn from, as it does not have any temporal context to help.
Therefore a different possible approach could be filming the running experiment or using a professional pedometer, in order to have the highest accuracy possible avoiding human and Android delays and inaccuracies in the recording process.

4.1.5.3 Long Short Term Memory Neural Network

To provide a simple explanation of the type of network that will be used a brief introduction over RNN must be made since they are the predecessors of LSTM.
It use the concept of *hidden state* (h_t), introduced in the *Hidden Markov model* to hold information on previously seen data and act as the NN memory.
To understand how the (h_t) is calculated it is possible to refer to Fig. 4.10 observing how the previous (h_{t-1}) is combined with the *current input* (x_t) to be passed through

the *tanh activation function*, resulting in the (h_t) to be passed to the next cell.

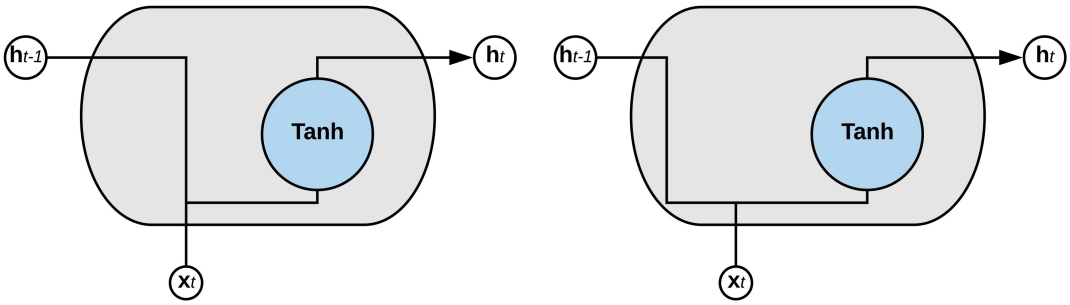


Figure 4.10: Structure of consecutive cells in a RNN

The just mentioned *tanh function* regulates the flow through the network, avoiding that values that undergo through many transformations reach astronomical levels, resulting unbalanced compared to new values.

$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

This is achieved "squashing" them into a range $\tanh(x) \in (-1, 1)$. It can be seen how the cellular structure is quite simple compared to the LSTM one in Fig. 4.11; this is because this last one is also provided with the ability to forget information when not considerate useful.
LSTM networks introduce two new concepts:

- the *cell state* that acts as the memory of network, transferring information along the sequence chain, allowing it to have a long term memory.
- *gates*, that decides which information is passed to the cell state; Therefore, during training, it will learn which informations are worth to remember or forget.

Here it must be introduced the *sigmoid function*, i.e. what allows the cell to decide that.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

This will produce a result in range $\sigma(x) \in (0, 1)$, it is indeed through that how values are forgotten, as any number multiplied by 0 results 0, causing it to vanish and

oppositely by 1 remain the same then it persists and is remembered.
Focusing now on the new elements introduced in the cells, it is possible to see four of them.

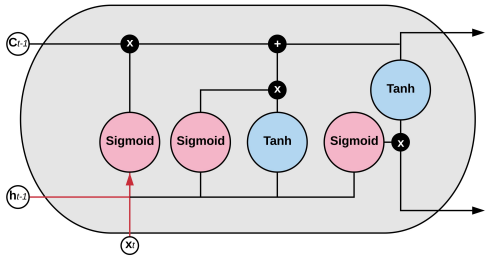


Figure 4.11: Forget Gate of a LSTM cell

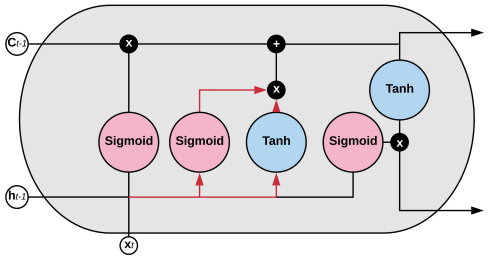


Figure 4.12: Input Gate of a LSTM cell

As highlighted in Fig. 4.11 the first element ìs the **forget gate**, the cell indeed receive information from the previous *hidden state* (h_{t-1}) and from the *current input* (x_t) and after combining them to form a vector apply to it the *sigmoid function*, determining the *forget rate output*(f_t).

The next step encountered in Fig. 4.12 is the **input gate**, the vector composed by (h_{t-1}) and (x_t) is passed into the *sigmoid function* that decides which values will be updated. The same vector $[(h_{t-1}) + (x_t)]$ is also passed into the *tanh function*, helping the regulate the network. Finally the two output are multiplied, allowing the sigmoid's output to establish which information from the tanh are considered relevant and will be remembered. The results will then be the *input gate output*(i_t).

It can be then observed in Fig. 4.13 how is calculated the **cell state**, firstly the *previous cell state*(c_{t-1}) is multiplied by the (f_t), allowing to drop values if the forget rate is close to 0. The obtained results is then summed with the (i_t) updating the *new cell state*(c_t) with values considerate relevant.

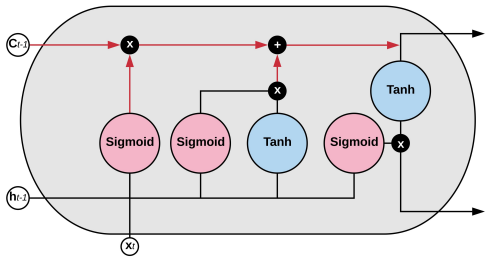


Figure 4.13: Cell State of a LSTM cell

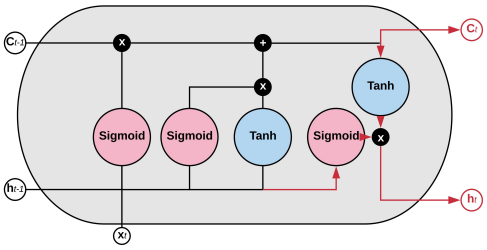


Figure 4.14: Output Gate of a LSTM cell

Finally the last phase is actuated by multiplying the result of (h_{t-1}) and (x_t) passed into the *sigmoid function* with (c_t) passed in the *tanh function*, obtaining the *hidden state*$(_t)$ of the current cell, that together with the (c_t) will be transmitted to the next cell.

Whit now in mind how a cell is composed it will be now possible to introduce the concept of *Layer*, that, as far as could have been investigated, in the *Keras* used framework differs a bit from the literature's definition, indeed it indicates a number of parallel and identical LSTM cells, although each eventually "learning to remember" different thing.

Creating the first layer then it is expected to be specified its dimension in the form of [*time steps, features*] these will must be then reflected in the number of *units*(official Keras documentation term), that for simplicity purpose may be regarded as cells, in the layer.

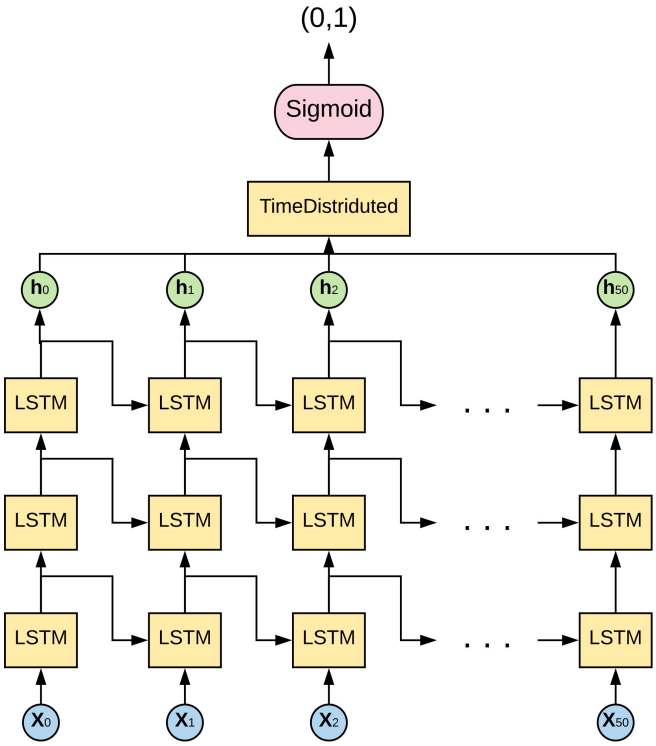


Figure 4.15: LSTM Architecture for Step Detection

It can be indeed seen in Fig. 4.15 each cell is passing its (h_t) to the cell at its right, affecting its (c_t), but also is used as input to the following layer.
This particular behaviour it is activated by the parameter `return_sequence=True`, differently if setted to `False`, as it is shown in Fig. 4.16, is just returned the (h_t) of the final cell of the layer.

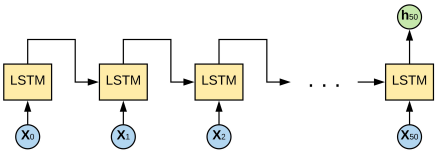


Figure 4.16: LSTM layer with return_sequence=False

This is not the desired behaviour in this case, as the succession of layers is used to increase the power of the network, increasing its ability to learn layer after layer.
Moreover, this also allows to use after the 3 LSTM layers a `TimeDistributed` layer, this will predict one value per *time step* considering the full sequence provided as

input. This means that, as shown in Fig. 4.15, all the 50 considerate *time steps* will influence the prediction of each of them, e.g. if there is an overall increase of the values of the provided sensors to start then decreasing it is highly likeable that it will be a step.
The 50 value has been chosen considering the averaged dataset created, as 50 consecutive *time steps* believed an optimal sequence length to produce predictions.
it must furthermore be mentioned that the `activation='tanh'` and *recurrent_activation='hard_sigmoid'* of layers hasn't been modified, as after some testing over `relu` and `PReLU`, resulted the ones to with better performances.

Finally it has been assigned to the `TimeDistributed` layer a *sigmoid activation function* to produce an output between 0 and 1, since our predictions will be binary value.
This model has then been compiled using `binary_crossentropy` to compute the cross-entropy loss between true labels and predicted labels, `Adam` optimiser and metrics `accuracy`. Again here has been tested different parameter on the optimiser, particularly *learning rate*, increasing it and *decay*, these two values are related as former influence latter as:

$$lr = initial\_lr * \frac{1}{1 + decay * iteration}$$

The methodology used here has been increasing the *learning rate* to have a quick initial learning, then when it would stop improving, stalling over a percentage of accuracy, the training would be shut down and have saved the best weights for each epoch, then load the best-performing ones and restart training using a lower *learning rate* and adding a *decay* to slowly after each epoch reduce the *learning rate* obtaining always finer results.
The model using the *Keras*'s provided method `predict_classes()` will be then able to pick the output values, as mentioned, in a range between 0 and 1 and produce a one-dimensional array of exactly 0s and 1s.

To be finally able to fit data on this model, it is necessary to reshape the

dataset created to fit the structure of the network, to do that it is useful the library *numPy* with he method `reshape`, indicating as the 3 new desired dimension: `(original_dataset_len/time_steps, time_steps, number_of_features)` This section will be concluded mentioning how, differently from the previously explained SVM model, it has been possible to apply weights, particularly creating them as mentioned with *Scikit-learn* using the method `compute_sample_weight()`, obtaining a one-dimensional array of values. Each of them corresponds to the weight calculation of the corresponding sample. This obtained array has been then reshaped as happened with the samples and fitted alongside the samples to the model.

4.1.5.4 Long Short Term Memory Neural Network Results

In this section, it will be provided an overview of the different approaches attempted and the consequent results.

Considering how the network will require to be trained several times over the same dataset to learn from it, it must be reckoned how the time required for the process will likewise rise too.

Therefore some preliminary evaluation has been made to be then able to proceed just in one direction.

The first question to be addressed was which dataset use between the one averaged over 3 or 5 time steps, hence has been tested their accuracy over 100 *epochs* (number of time that the training dataset is analysed by the algorithm).

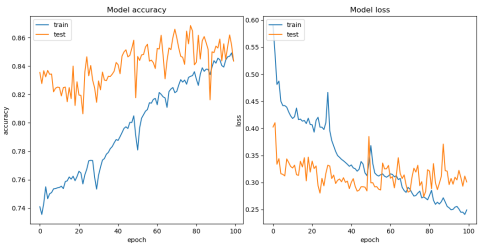


Figure 4.17: Accuracy and Binary Crossentropy loss development over training on 100 epochs with dataset averaged over 5 time steps

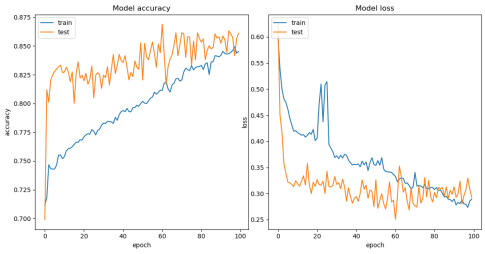


Figure 4.18: Accuracy and Binary Crossentropy loss development over training on 100 epochs with dataset averaged over 3 time steps

Although not being that dissimilar, observing the differences in*accuracy* and Binary

Crossentropy logarithmic loss visible in Fig. 4.17 and Fig. 4.18, it can be noticed how training over the dataset averaged with a mean value of 3 showed a more linear and less jittery learning curve in the accuracy over both training and validation data, seeming more promising to further investigation.
This dataset has then been used again for the training, this time over 300 epochs in Fig. 4.19 and then since it has not shown sign of overfitting or major stops in learning, the saved best weights has been loaded again and trained other 300 times.

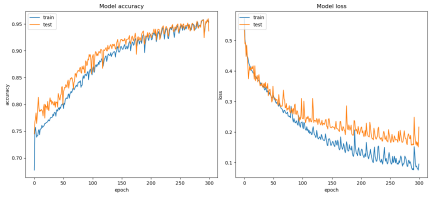


Figure 4.19: Accuracy and Binary Crossentropy loss development over training on 300 epochs

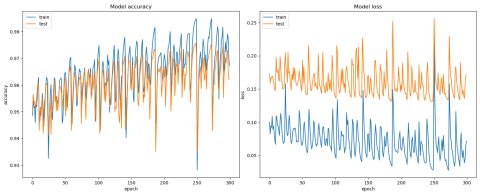


Figure 4.20: Accuracy and Binary Crossentropy loss development over training on other 300 epochs

It can be noticed in Fig. 4.20 how although the growth seems more jittery, it is mostly due to the different scale of the two plots. What is interesting is how, albeit if slowed the learning phase did not seem to be stopped, suggesting that with more training, even better results could be achieved. However the time restriction of the project did not allow further analysis to test this theory, but these clues highly suggest it.

It must be specified that these results have been achieved using *Keras* default *batch size* of 32 as seemed an opportune enough length of the sequence to be observed.
To now define what the batch size is, it can be expressed as the number of inputted samples after which the state of the model is reset and hence its gradient and weights or parameters updated, increasing the batch size would mean lengthen the backpropagation, having a better gradient estimation and prediction.

Moreover, the obtained optimal weights founded after this training has then been tested using as final layer a `TimeDistributed` one, as previous mentioned, as well as with just a normal `Dense` one, to test the difference in performances between the two.

	Prediction over Validation Dataset				
	Steps	**Predicted**	**Accuracy**	**Precision**	**Recall**
LSTM Dense final layer	2539	5250	0.99	0.05	0.10
LSTM TimeDistributed final layer	2539	9711	0.98	0.04	0.17
SVM	2539	78277	0.75	0.02	0.75

Table 4.11: Prediction results final comparison between tested algorithms

As can be observed in Tab. 4.11 surprisingly similar and even better results has been achieved using the *Dense* layer, this could be probably addressed to the previously discussed learning mechanism of each cell and how the previous context is already considered with *forget* and *input gate*, therefore it could be that this is already the right amount of information needed and having to decide over each data-point considering the 50 provided time steps could result overwhelming.

It can also be noticed in Fig. 4.21 how this has been achieved with a training over 10 epochs, and after an expected initial adjustment there is an organic growth, indicating how there is still margin for optimization over these weights, it is indeed likely that with more training will further improve reaching results that are reasonably acceptable for the project's purposes.

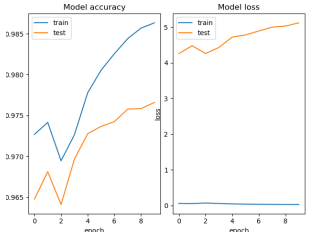


Figure 4.21: Learning curve obtained using a Dense layer as final one

As a final analysis, it can be observed in Fig. 4.22 how the surplus detected steps are located compared to the manually detected ones, it is interesting to notice how instead of being predicted randomly they are grouped around the actual steps.

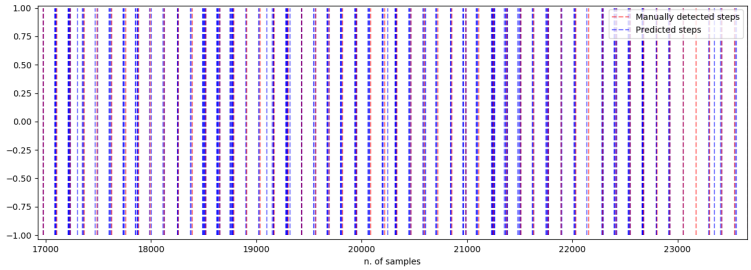


Figure 4.22: Comaprison between placements of manually detected and predicted steps

Seeing this it is possible to formulate two observation: firstly, the initial belief that the model could use the available context, even as a sort of autocorrelation function to predict where the next possible beat will be, could not be entirely accurate; since more than one are predicted where there should be just one. Secondly, it means that to improve prediction accuracy, it will be simply possible to don't use the *Keras* provided method `predict_classes`, but instead `predict`. This latter will return the probability of a step in a range between 0 and 1 to then apply to these results another algorithm that will check if nearby there are already predicted steps and if so keeps the one with the highest probability.
This should reduce the number of multiply detection for the same step, leaving just the problem of the not predicted steps, as can be seen on the extreme right of Fig. 4.22.

4.2 Beats Detection and Counting

In this section will be introduced and examined the beat detection problem, enriching with new analysis the proposed knowledge regarding LSTM networks.

4.2.1 Experimental Set-Up

In order to have a diverse and appropriate music dataset, it has been decided to do not use the few available ones. It has been instead decided to develop a new one to be able to select songs that could be considerate contemporary and "motivational", therefore likely be as close as possible to what could possibly be in the final user playlist.
This choice is justified by the fact that most of the dataset available have different

features labelled. Prevalent examples are music genre or mood of the songs and some even total BPMs, but it has been hard to find something where each beat is labelled. Furthermore, some datasets do not provide the actual audio files, but just the extracted features, while it was the interest of this project to explore this process as it would be required to be performed over the final user's playlist. Therefore has founded as an optimal solution the use of music downloaded under creative commons license, being also extra careful to select only songs that did not require attribution of any credits. Moreover, the files will not be made available, distribute or reproduced anywhere.
This allowed choosing music manly between the rock, hip-hop, electronic and pop genres, in accordance with preferences extracted in the questionnaire carried out, visible in Fig. A.5.

4.2.2 Data Preparation

All the downloaded songs have been collected on a specific directory, however, all of them were of *mp3* file format, hence it has been necessary to write a Python script that uses the **"ffmpeg"** framework to convert each of them into the more usable *wav* format and place them on a new folder.

4.2.3 Data Analysis and Preprocessing

From this point forward it will be most useful the Python library **LibROSA** that allow loading *wav* files and perform many feature extraction, beat and tempo detection and plotting functions essential to this project. The collected files have been loaded on Python through `librosa.core.load()`, this method allows to select some parameters in this process, particularly has been used `res_type='kaiser_fast'` (it not use the default best quality resampling mode in favour of a superior speed) and `sr=22050` (it define the target sampling rate). These two choices have been made to reduce the time required by the process since it is not required high quality, as actually, it is better to lower it to reduce the incredible amount oh samples. Moreover, by sample rate it is meant the number of samples recorded in a second and usually, songs collected included, is 44100. It has been decided to halve this number as, after some attempts,

resulting in the lowest possible without then losing accuracy on beats detections.
This is indeed the main goal here and has been achieved in a three steps process.
The first thing has been to find the track's BPM and to do that has been used the very popular DJ software **Traktor**, that has between his best features an automatic and quite reliable BPM's recognition tool.
Secondly has been used the provided method `librosa.beat.beat_track`. This detect beats with the following methodology:

1. measuring the onset strength,
2. estimating the tempo from the onset correlation,
3. picking peaks in onset strength approximately consistent with estimated tempo. [19]

This will return the predicted BPMs and beats location in the specified units(frames, samples or time).
To check the accuracy of the estimated tempo, it has been compared with the results obtained with Traktor and, as can be seen in Tab. 4.12, although being acceptable in most of the cases, still some inaccuracy exist.

TraktorBPMs	102,000	107,997	108,000	110,000	111,000	119,999	120,004	123,996	124,500	125,041
LibrosaBPMs	103,359	107,666	107,666	112,347	112,347	117,454	117,454	123,047	83,354	123,047
TraktorBPMs	127,349	128,000	130,000	131,999	132,000	139,751	148,001	153,998	80,000	87,490
LibrosaBPMs	129,199	129,199	129,199	129,199	89,103	143,555	151,999	151,999	161,499	117,454

Table 4.12: Comparison between Traktor and LibROSA detected BPMs

However, to overcome that, `librosa.beat.beat_track` also provides the possibility to pass as a parameter(`start_bpm=`), i.e. the previously known BPM if available and hence it was inserted here the values obtained with *Traktor*, in order to increase the accuracy.
The final results have been then manually analysed to check the precision of predictions. This was performed firstly by ear, counting if the number of beats was reasonable and when matching between Kick drum and beats was linear through plots analysis.

This is evident in Fig. 4.23, where the peaks of amplitude caused by the high energy produced by the kick drum are nicely matched with the beats.

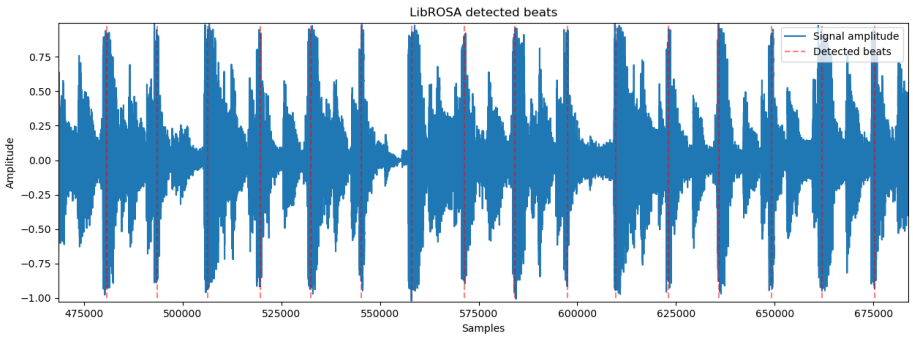


Figure 4.23: LibROSA Detected Beats

Since `librosa.beat.beat_track` returns the index of samples where a beat is estimated this had to be converted in an array of zeros and ones to then be used as labels:

```
def gen_labels(beats, lenght):
        for e in range(beats[1], lenght):
        if e in beats:
        yield 1
        else:
        yield 0
```

Listing 4.4: Conversion of LibROSA detected beats into binary labels

This method would then just rwquired to be called inside a **list** to be able to have a dataset of the desired labels.

The next step of investigation has been the feature extraction, and mainly three option has been evaluated:

- **STFT** differently from the standard Fourier transform, that return information about the frequency of a signal averaging them over the whole length, STFT present time-localized frequency information calculating the FFT on a window that is moving all along the signal length;

- **mel spectrogram**, to determine it is first calculated the magnitude of its spectrogram, that then is mapped using the *Mel scale*;
- **Revisited Moving Average**, very similar to the one applied in the data preprocessing for step detection, but with different objectives and `mean_value`.

Now a graphic representation will be provided in order to have a better visualization of these concepts and allow an easier comprehension of the mental process followed to arrive at a final decision on which one could be the optimal feature to be fit in the NN.

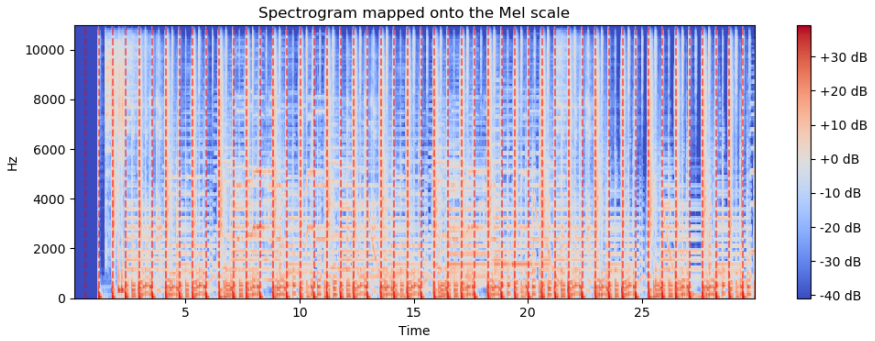


Figure 4.24: Spectrogram mapped onto the Mel scale compared with LibROSA Detected Beats

As can be seen in Fig. 4.24 the *mel spectrogram* has been applied over 30 seconds of a song, and precise and reliable indications over the positioning of the beats can not easily be evinced, not surprisingly indeed has offered its best results when used as a mere tool for classification instead to be analysed as a time sequence. This can be better exemplified providing the example of the *music genre recognition* or *speech recognition problem*, in this case indeed the particular characteristics of diverse genres or sounds can be reflected in a recognisable spectrogram.

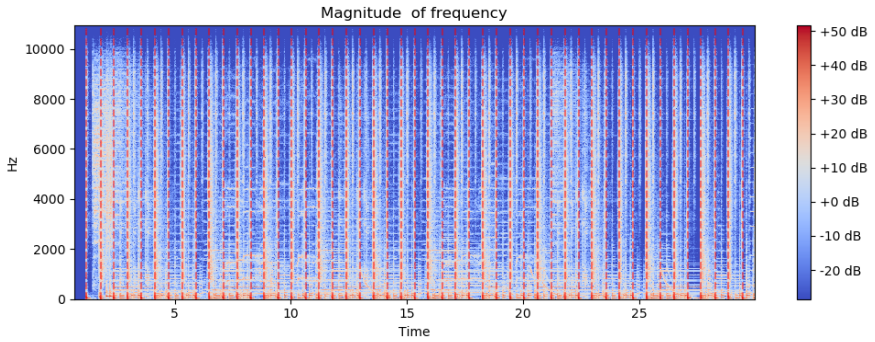


Figure 4.25: Magnitude of frequency calculated with STFT compared with LibROSA Detected Beats

Even if in Fig. 4.25 a more readable output is produced, however the same argument can be reapplied here. Furthermore, these methods add complexity in the labelling of the data, since it is straightforward to apply a label as music genre over the spectrogram of a song or segment of it, different instead is having to apply labels exactly where several beats are. This will be clearer providing as example the analysis of a random song picked from the dataset.

It is has a shape of a one-dimensional array of 4137618 samples; it is, therefore, a straightforward process to assign labels to each sample, on the contrary, the *STFT* will produce a bidimensional array of size (1025, 8082) and the *melspectrogram* (128, 8082).

This means that the labels produced, of the same form of the samples, must be converted into these formats, and time constraints of this project did not allow to investigate the subject further.

Therefore the most appropriate solution seemed to apply the *Revisited Moving Average* since it allows to achieve simultaneously two interesting goals. Firstly to reduce the signal complexity to have more meaningful data to fit in the ML model. Secondly to reduce the number of samples in order to decrease the computational power and memory required to train the model, since the available GPU (*Nvidia GeForce GTX 1060*) was struggling to produce result otherwise.

Not to mention that, as can be imagined, the database is again heavily unbalanced, therefore reducing that it is another welcomed consequence.

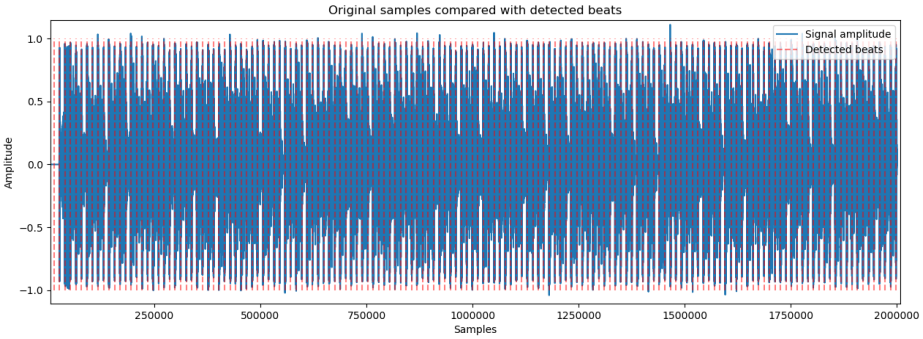


Figure 4.26: Original samples compared with Detected Beats

Both of these goals can be observed analysing a minute of a random song, comparing the original signal in Fig. 4.26 too noisy and complex, moreover with more than 1 000 000 samples, compared to Fig. 4.27 where the peeks nicely match with the detected beats and just 600 samples.

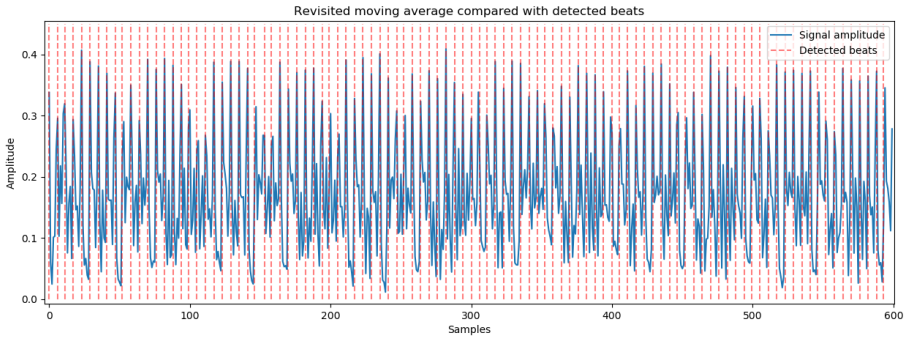


Figure 4.27: Original samples compared with Detected Beats

It has been probably noticed how to a minute of song corresponds exactly 600 samples, this is because as a `mean_value` in the moving average has been chosen 2205, i.e. $\frac{1}{10}$ of the selected sampling rate.

This becomes particularly useful in the next step of the data preprocessing, as each song, according to his length has been trimmed from the second beat detected (the first one usually happens before than the music, and particularly the kick drum, properly starts) to a multiple of a minute.

Having then the audio segment all a fixed and known size, it is hence possible to take

advantage of the remembering abilities of RNN. Indeed it is possible to fit it with a defined number of *time steps* such that it will learn just from a song, avoiding the sudden variation in the tempo between two songs, that could result confusing.
Again, as mentioned, to complete the balancing of the dataset has been required the use of appositely created weights. It has been then applied the same *scikit-learn* method as for the step detector.

Since the data preprocessing process it requires some time it has then been created a final dataset on *CSV* files, containing one for each row on of the 45 trimmed in length songs to which it has been applied the *Revisited Average*.

4.2.4 Data Classification and Training

In the classification process, due to the increased complexity of the problem compared to the previous one, it has manly considerate NN. Mainly two option: similarly as happened with the step counter an LSTM on averaged results, or a CNN over STFT and Mel spectrograms produced data, since these produce a 2D output that will fit well on CNN.
However, after some experimentation on the second option has been chosen to go with the first one. This has been mainly motivated by the time limitation and the assumption that CNN could be more appropriate to genre recognition, rather than for the examined problem. Furthermore, it has been preferred to furtherly investigate the various potentiality of different configuration in a LSTM network increasing the level of confidence in the understanding of how data are processed during the training and therefore produce better results for both the treated problems.
Therefore three different configurations of the network have been explored to test its remembering and prediction abilities.
To start this investigation has been selected the network structure visible in Fig. 4.28 selecting a layer with 200 cells, therefore corresponding to 200 samples, which in turn, considering the chosen sampling rate are exactly 20 seconds of a song.
This unit of measurement has been chosen to provide a meaningful length to be anal-

ysed, from which it is possible to understand and extract the beats having enough context to learn from. The hope here has been that information about the repetitiveness of beats can be transmitted along the cells' chain in the layer, providing the next cells with more context about the probability of when could be the next beat.
Moreover, it has been chosen to use a single layer to reduce the training time, reducing the complexity of the network, to then continue with further investigations only on the best performing one.

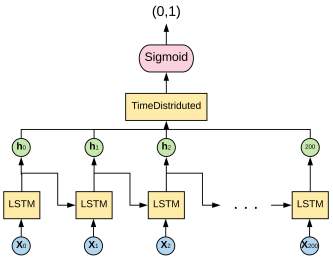


Figure 4.28: LSTM structure for BPM recognition

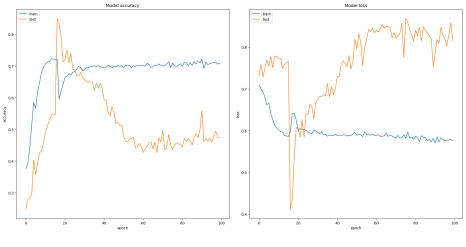


Figure 4.29: Learning curve obtained using a TimeDistributed final layer, making the network stateful

The first approach attempted has been to expose the network to the entire sequence, hoping that it could autonomously learn inter-dependencies, rather than having to fit it with data organized, as mentioned, in a way that it will learn just from a single song. This can be achieved using the *Keras* framework by making the LSTM layer `stateful`. This means that the state of the network will not be reset after the provided sequence of *time steps* or defined *batch size*, but instead, it will require to be manually reset after every epoch. This is the proper way to take full advantage of the LSTM potentiality. Indeed all the data in training dataset will be considered during prediction.
Nevertheless, despite this the results of training over 100 epochs, shown in Fig. 4.29, are quite disappointing.
It is indeed shown a clear case of *overfitting*, i.e. the model concerned learns too in details data and noise in the training dataset, impacting negatively on its ability to generalize and therefore predict over different data.
This can be noticed in how the learning progress visible on the training data are not reflected over the validation ones and how oppositely the binary cross-entropy loss while decreasing in training has an opposite behaviour on the validation.

This could be probably imputed to the relatively small size of the dataset, as it has been trained over just 45 songs, using a relatively high averaging mean value of 2205 that reduces the number of data-point available significantly.
Moreover, this implementation could not be too well suited to this issue, since although it is not really a time sequence, as indeed a series of different consecutive sequences. Therefore the transition between songs and hence diverse tempo could be learned as a concept from the model, producing the mentioned overfitting. This would indeed be more indicated to pure time sequences as it could be the case of stocks in financial markets.

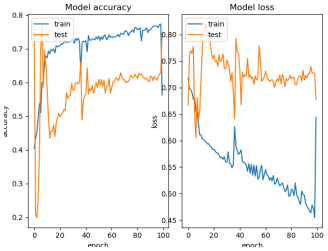


Figure 4.30: Learning curve obtained using a TimeDistributed final layer without shuffling samples

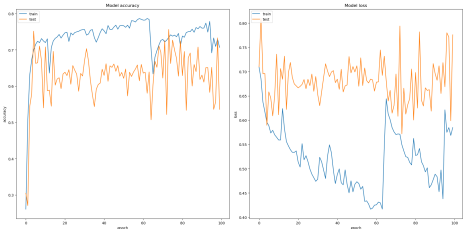


Figure 4.31: Learning curve obtained using a TimeDistributed final layer shuffling samples after each epoch

Hence two different approaches has then been tested to treat differently this problem again over 100 epochs:

- firstly, considering how in a non-stateful LSTM layer the state is reset after each batch it has hence considered to use a batch size of 3, therefore, since the network is fed with sequences of 200 samples, it will learn considering 3 of these. However, this context will not be available when it produces prediction, as only the inputted sequence is used. Summarising the model will learn considering 60 seconds of a song and predict over 20.

- secondly, has been tested the dataset as a group of 20 seconds length input, hence maintaining the same 200 input shape, but reducing the batch size to 1 and after each epoch randomly shuffling these sequences. In doing so, the model, will reset its state after each provided input learning and predicting over 20 seconds but will be trained with a dataset reordered each time resulting as new to it. This

was an attempt to completely avoid cases of the aforementioned overfitting .

However, neither of the results produced by the former in Fig. 4.30 nor by the latter in Fig. 4.31 has been particularly promising. Both have indeed shown how the learning curves faced a drastic drop closer to an accuracy of 80%. The reason for this has been again imputed to the dimension of the dataset, not being big enough. Therefore other 45 songs have been included to approximately double its size.

The new dataset has then been tested using a more robust network, to see if rising the number of layer to 3, as shown in Fig. 4.32, it is possible to increase the learning ability of it.

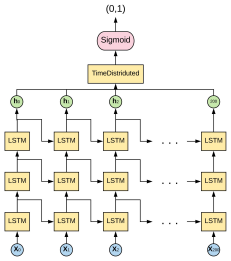


Figure 4.32: Model structure composed by 3 LSTM layers and a TimeDistributed final layer

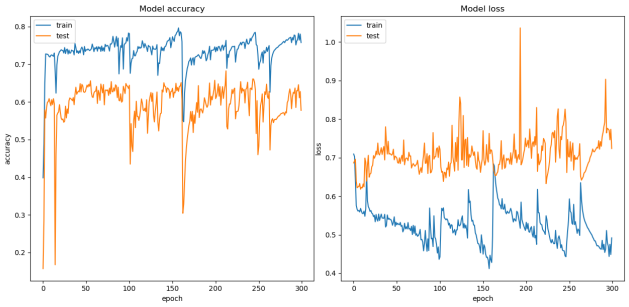


Figure 4.33: Learning curve obtained using a 3 LSTM layers without shuffling samples over 300 epochs

Unfortunately, as can be observed in Fig. 4.33, even over a major number of epochs, it is not yet sufficient to significantly improve its learning abilities. It can be noticed how still the 80% of accuracy over the training data is an insurmountable barrier, and furthermore, its ability to generalise is rather reduced, observing the substantial differences in accuracy between training and validation data. Therefore it can be imagined how the precision of the model would be very modest, as indeed just 21% over-predicting 15881 beats against the actual 5319 present in validation data.

It is, therefore, observing how has been challenging to achieve results fitting the model with well-defined sequences, bothering to reset the model state when there is a change in song or tempo. It has then been tested the model reaction being exposed to these variations observing if its ability to generalise could improve.

The batch size has been then increased to 20, and the model trained over other 100

epochs obtaining slightly better results with accuracy just above 80% and precision increased to 25%.
That confirms how the effort made to fit clean data did not pay, and sometimes more raw data can be better generalised, besides the fact that a batch size of a substantial dimension means learning from a more extended sequence of data and not to be underestimated considerably shorter training times.

Finally, over these considerations and remembering how the dataset has been considered too small, another approach has been attempted. In order to increase its size and at the same time test the network in a different but intriguing way, the dataset has been modified with the goal to fit the model, not with distinct and not overlapping sequences, but instead with moving windows.
Being not that simple to explain that in words it would be provided an example, hence considering a dataset hypothetically composed by [1,2,3,4,5,6,7,8,9] the first and until now applied method would be to fit the model with 3 samples, composed in this way [1,2,3], [4,5,6] and [7,8,9]. The new method applied here instead will fit [1,2,3], [2,3,4], [3,4,5], [4,5,6], [5,6,7], [6,7,8], [7,8,9].
It is evident how the number of samples provided would increase considerably and will allow analysing the same time step multiple time with different context available.

```
def gen_window_dataset(X, window=200):
        l = len(X) - window
        newX =[]
        for i in range(l):
        newEntry = df[i:i + window]
        newX.extend(newEntry)
        return newX
```

Listing 4.5: Used method to generate a windowed dataset

It is possible to see in Listing 4.5 how this has been achieved using a for loop iterating each data-point in the dataset and each time adding a window of the selected size to the newly created dataset. Then it would be just necessary to reshape the obtained list to a 2D *NumPy* array with as a first dimension the length of the dataset divided by the window size and as a second one the actual window size, as shown in Listing 4.6,

to have a dataset shaped as desired.

```
newX = newX.reshape(len(newX)//window, window, 1)
```

Listing 4.6: Used method to reshape the windowed dataset

In doing so, through a moving window of size 100, from a dataset shaped for training as (392, 200, 1) and for evaluation (169, 200, 1), the new one would result respectively of dimension (39235, 200, 1) and (16815, 200, 1).
This dataset was then used for training over 50 epochs again the same three-layer LSTM model, but using a batch size of 100 to include more context and reduce the computing time of a larger dataset.

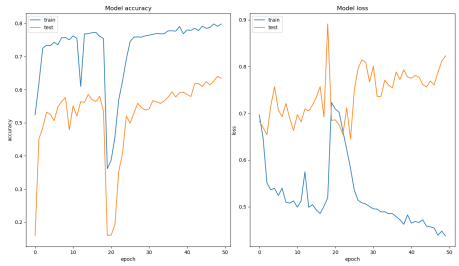


Figure 4.34: Learning curve obtained using 3 LSTM layers without shuffling samples using a windowed dataset over 50 epochs

It is possible to notice how in Fig. 4.34 even with this model accuracy of 80% seemed the maximal possible in training, again not well reflected over the validation data.
Moreover, the logarithmic loss seems to follow different trends in training and validation, showing sign of overfitting.
It is main possible that the validation data are notably different from the training data and dividing the dataset differently better results could be achieved.

Overall, it seems evident how this approach is not producing accurate results. This can be addressed to a lack of features provided to the model, in the step counter one it has been indeed observed how fitting six different features produced excellent results.
It is not easy here to figure it how to combine different features, but probably it would be required to consider some frequency or magnitude measurements such as the STFT

as in fact applied by [12] or maybe fitting the MFCC (computing the DCT over the mel-spectrogram) on a CNN.

Furthermore, another possible approach would be to consider the use of *bidirectional LSTM layer* as suggested in [18], this has quickly attempted, but again, the difficulty to fit weights has stopped further analysis.
It must hence be noticed how this, due to time restriction, it has not been fully experimented and probably implementing weights at a class level instead of at samples one could be implemented. This will require to modify the labels to transform them into a binary matrix with two columns, one of them representing *beats* and the other*non beats*.
It will be furthermore required to change the activation function of the final `TimeDistributed` layer from *sigmoid* to *softmax*. This is because, as mentioned, labels now are two columns; therefore, e a sigmoid function can not be applied anymore. However using a softmax activation on two classes will produce the same results, as they are mathematically equivalent.
The use of *bidirectional LSTM layers* will then allow the network to analyse al the fitted sequences of timesteps in both the direction allowing to have a much broader context without overloading the memory and then computational time.
This amendment could possibly lead to better performances and predictions.

4.3 Prototype Evaluation

As a final step of this research, it has been tested the produced prototype to evaluate its behaviour over the same condition to the data acquisition experiment, to have a comparison and evaluate how well the implementation of the model could fits in the produced prototype.
It has been mainly tested *availability* and *usabilIty* and the app, as far has it has been possible to test, resulted bug-free and pleasant to use.
This obviously is just the biased opinion of the researcher itself; however, it is still

noticeable how the songs are actually reproduced with the desired logic, and they are sensitive to different paces.

Chapter 5

Conclusion and Future Work

After having concluded as much as possible experimentation and discussed their implementation and results, it is now possible to conclude evaluating how well it has been achieved the objective of this dissertation.
Furthermore, the areas of possible improvements, as it was not possible to cover them through this dissertation, will be highlighted to encourage future works.

As one the main interest of this dissertation was to explore the whole process of developing required to build Neural Networks model, regarding the step detection and beat detection problems, it can be stated that it has been individuated methodologies capable of producing results.

In details, the approach proposed to step detection can be considerate a remarkable result, compared to the lack of resources and time.
It has indeed been achieved similar results to the Android's built-in step counter, with both detecting approximately the double of the actual steps.
Not to mention, how with the proposed Improvements, i.e. a further long training, use of more accurate methodologies in the collecting of the data and implementation of an algorithm to refine the model prediction, it would be ideally possible to obain better performaneces than th eones achieved by the Android's detector.
It can then be addressed to future work more test of how to implement SVM models

assigning weights at classes or samples level.
It was then proposed a suitable solution and an in-depth overview of how to record data from Android's accelerometer and gyroscope sensors whit the confidence that even more sensors could be recorded without notably raising the recording delay. This is achieved thanks to the combination of using a ThreadPool and multiple SensorEventListener.

Regarding the beat detection algorithm, the results have been instead disappointing, leading to questioning the approach attempted to follow a more canonical one.
It has been already proposed how the use of STFT or MFCC to fit CNN or bidirectional LSTM could probably lead to better results.
Nevertheless, it has to be admitted to how the design of a precise methodology to record steps, as well as the implementation of the relative model, required more times that estimated and therefore it has not been possible to evaluate as desired the beat detection problem.
Nonetheless, a practical methodology to identify beats and convert them to labels ready to be fitted in a model has been provided, obtained combining LibROSA Python's library and the software Traktor.

Further study should also be deserved to test how it would be possible to have a matching not between BPM and steps per minute, but at a single beat-step level. However, to be able to accomplish that it would be before accomplished an accurate detection of both.

Finally, it has been proved possible to match user running pace with a song's BPM, realizing a fully functional prototype, to which in future work would be interesting to implement in it the developed ML algorithms and test if and how the smartphone hardware's lower specs can handle them.
Moreover, the proposed algorithms have to be tested on live scenarios to check if they replicate the same performances proposed.

Appendix A

Figures

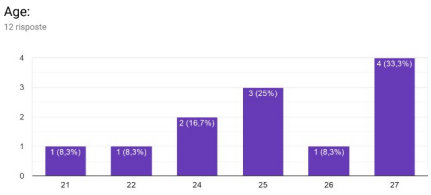


Figure A.1: Age Distribution

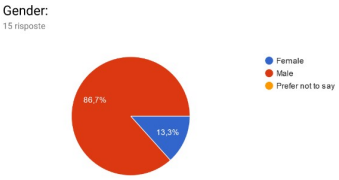


Figure A.2: Gender Distribution

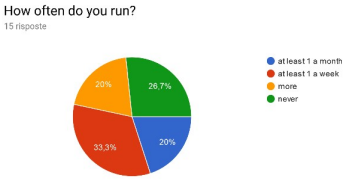


Figure A.3: Run Frequency

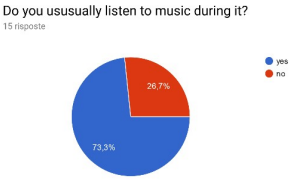


Figure A.4: Music Frequency

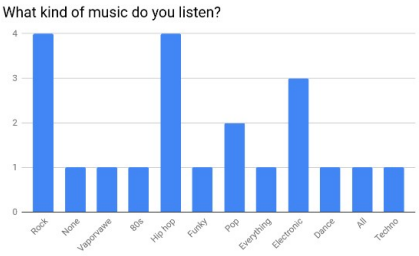


Figure A.5: Music Genre Distribution

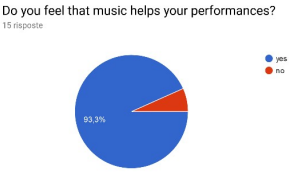


Figure A.6: Perceived boost in Performances

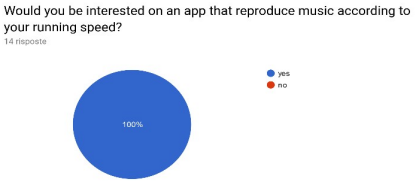


Figure A.7: Expressed Interest in the App

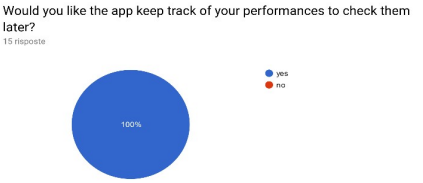


Figure A.8: Expressed Interest in Performances Tracking

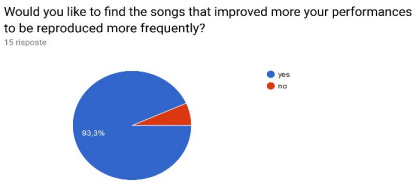


Figure A.9: Interest in Reproducing more often Songs that Leads to better Performances

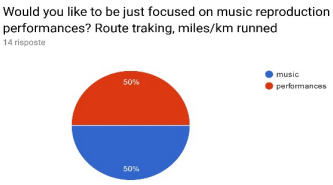


Figure A.10: Focus on Music or Performances Tracking

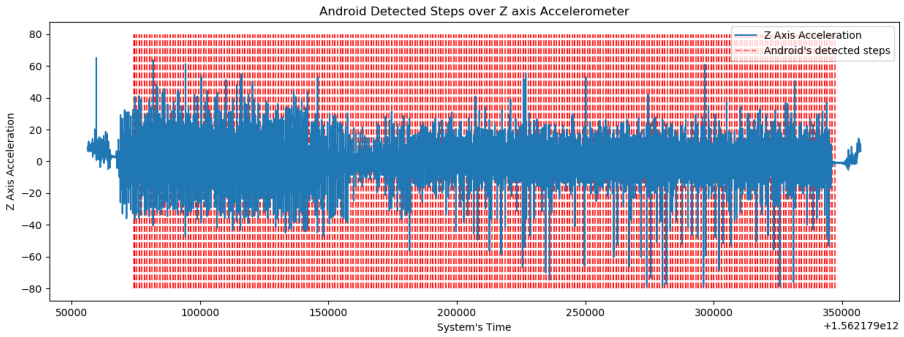
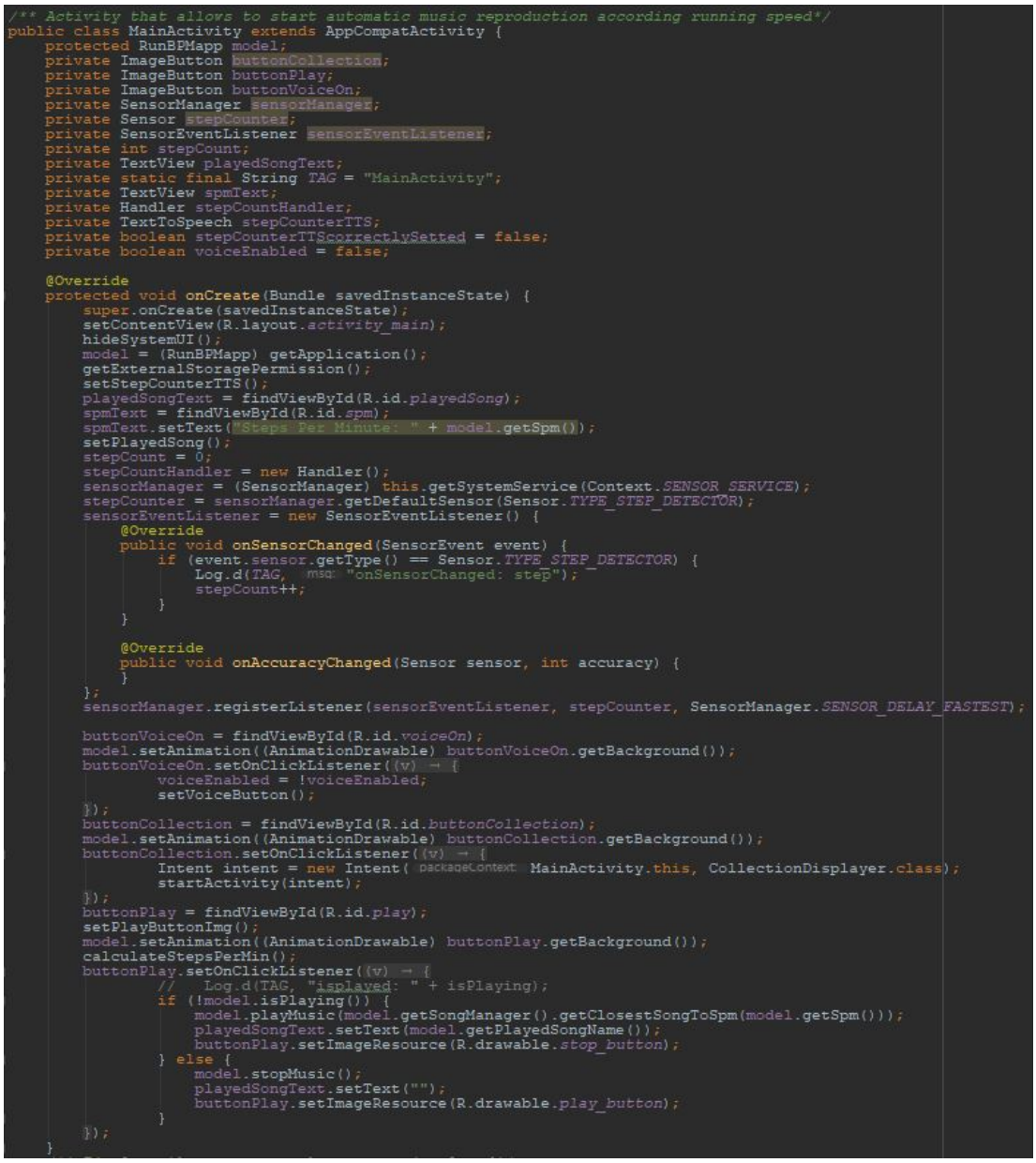


Figure A.11: Android Detected Steps over Z axis Accelerometer (5 min. approx.)



```
/** Activity that allows to start automatic music reproduction according running speed*/
public class MainActivity extends AppCompatActivity {
    protected RunBPMapp model;
    private ImageButton buttonCollection;
    private ImageButton buttonPlay;
    private ImageButton buttonVoiceOn;
    private SensorManager sensorManager;
    private Sensor stepCounter;
    private SensorEventListener sensorEventListener;
    private int stepCount;
    private TextView playedSongText;
    private static final String TAG = "MainActivity";
    private TextView spmText;
    private Handler stepCountHandler;
    private TextToSpeech stepCounterTTS;
    private boolean stepCounterTTScorrectlySetted = false;
    private boolean voiceEnabled = false;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        hideSystemUI();
        model = (RunBPMapp) getApplication();
        getExternalStoragePermission();
        setStepCounterTTS();
        playedSongText = findViewById(R.id.playedSong);
        spmText = findViewById(R.id.spm);
        spmText.setText("Steps Per Minute: " + model.getSpm());
        setPlayedSong();
        stepCount = 0;
        stepCountHandler = new Handler();
        sensorManager = (SensorManager) this.getSystemService(Context.SENSOR_SERVICE);
        stepCounter = sensorManager.getDefaultSensor(Sensor.TYPE_STEP_DETECTOR);
        sensorEventListener = new SensorEventListener() {
            @Override
            public void onSensorChanged(SensorEvent event) {
                if (event.sensor.getType() == Sensor.TYPE_STEP_DETECTOR) {
                    Log.d(TAG, msg: "onSensorChanged: step");
                    stepCount++;
                }
            }

            @Override
            public void onAccuracyChanged(Sensor sensor, int accuracy) {
            }
        };
        sensorManager.registerListener(sensorEventListener, stepCounter, SensorManager.SENSOR_DELAY_FASTEST);

        buttonVoiceOn = findViewById(R.id.voiceOn);
        model.setAnimation((AnimationDrawable) buttonVoiceOn.getBackground());
        buttonVoiceOn.setOnClickListener((v) -> {
                voiceEnabled = !voiceEnabled;
                setVoiceButton();
        });
        buttonCollection = findViewById(R.id.buttonCollection);
        model.setAnimation((AnimationDrawable) buttonCollection.getBackground());
        buttonCollection.setOnClickListener((v) -> {
                Intent intent = new Intent( packageContext: MainActivity.this, CollectionDisplayer.class);
                startActivity(intent);
        });
        buttonPlay = findViewById(R.id.play);
        setPlayButtonImg();
        model.setAnimation((AnimationDrawable) buttonPlay.getBackground());
        calculateStepsPerMin();
        buttonPlay.setOnClickListener((v) -> {
                //   Log.d(TAG, "isplayed: " + isPlaying);
                if (!model.isPlaying()) {
                    model.playMusic(model.getSongManager().getClosestSongToSpm(model.getSpm()));
                    playedSongText.setText(model.getPlayedSongName());
                    buttonPlay.setImageResource(R.drawable.stop_button);
                } else {
                    model.stopMusic();
                    playedSongText.setText("");
                    buttonPlay.setImageResource(R.drawable.play_button);
                }
        });
    }
```

Figure A.12: First part of MainActivity class

```
    /** Notices the user with his running speed (teps per min.)*/
    private void pronounceSPM() {
        if(voiceEnabled) {
            String text = "your running speed is " + model.getSpm() + "steps per minute";
            stepCounterTTS.speak(text, TextToSpeech.QUEUE_FLUSH,  params: null);
        }
    }

    /**Hides navigation bar*/
    private void hideSystemUI() {
        // Set the IMMERSIVE flag.
        // Set the content to appear under the system bars so that the content
        // doesn't resize when the system bars hide and show.
        getWindow().getDecorView().setSystemUiVisibility(View.SYSTEM_UI_FLAG_LAYOUT_STABLE |
                View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
                | View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN
                | View.SYSTEM_UI_FLAG_HIDE_NAVIGATION // hide nav bar
                | View.SYSTEM_UI_FLAG_FULLSCREEN // hide status bar
                | View.SYSTEM_UI_FLAG_IMMERSIVE);
    }

    /** Refreshes the GUI of this activity*/
    @Override
    public void onResume() {
        super.onResume();
        setPlayButtonImg();
        setPlayedSong();
    }

    /** Terminates TextToSpeech and Runnable object at app closure*/
    @Override
    protected void onDestroy() {
        if (stepCounterTTS != null) {
            stepCounterTTS.stop();
            stepCounterTTS.shutdown();
        }
        if (stepCountHandler != null){
            stepCountHandler.removeCallbacks(stepCountRunnable);
        }
        super.onDestroy();
    }

}
```

Figure A.13: Third part of MainActivity class

```java
/** Class that represent a song */
public class Song implements Serializable {
    private File file;
    private String name;
    private int BPM;

    /** Class constructor
     * @param file song's path
     */
    public Song(File file) {
        this.file = file;
        name = file.getName();
        autoStetBPM();
    }

    /** Returns song's path
     * @return File
     */
    public File getFile() { return file; }

    /** Returns song's name
     * @return String
     */
    public String getName() { return name; }

    /** Returns song's BPM
     * @return int
     */
    public int getBPM() { return BPM; }

    /** Automatically sets song's BPM from the song's title.
     * !!! ASSUMPTION: All the audio in the device have their BPM as title.
     * This Will change implementing the ML algorithm to detect BPM
     */
    public void autoStetBPM () {
        BPM = Math.round(Float.valueOf(name.substring(0, name.lastIndexOf( str: "."))));
        System.out.println("bpm: " + String.valueOf(BPM));
    }

}
```

Figure A.14: Complete Song class

```java
/** Adapter that allows to display the song collection*/
public class SongAdapter extends RecyclerView.Adapter<SongAdapter.ViewHolder> {
    private ArrayList<Song> songCollection;
    private OnNoteListener onNoteListener;
    /** ViewHolder inner class*/
    public static class ViewHolder extends RecyclerView.ViewHolder implements View.OnClickListener {
        public TextView songNameView;
        public OnNoteListener onNoteListener;
        /** ViewHolder constructor
         * @param itemView
         * @param onNoteListener
         */
        public ViewHolder(View itemView, OnNoteListener onNoteListener) {
            super(itemView);
            songNameView= itemView.findViewById(R.id.songName);
            this.onNoteListener = onNoteListener;
            itemView.setOnClickListener(this);
        }
        /** Calls the onNoteClick method on the clicked item
         * @param v the View
         * */
        @Override
        public void onClick(View v) {
            onNoteListener.onNoteClick(getAdapterPosition());
        }
    }
    /** Constructor for the SongAdapter class*/
    public SongAdapter(ArrayList<Song> songCollection, OnNoteListener onNoteListener){
        this.songCollection = songCollection;
        this.onNoteListener = onNoteListener;
    }

    /** Creates the ViewHolder and place it inside the current ViewGroup
     * @param viewGroup
     * @param i the viewType which the holder should be placed in
     * @return the viewHolder
     */
    @NonNull
    @Override
    public ViewHolder onCreateViewHolder(@NonNull ViewGroup viewGroup, int i) {
        View v = LayoutInflater.from(viewGroup.getContext()).inflate(R.layout.song_item, viewGroup, attachToRoot: false);
        ViewHolder vh = new ViewHolder(v, onNoteListener);
        return vh;
    }
    /** Sets the item from the list into a specific holder in the recyclerView
     * @param viewHolder
     * @param i the position within the list
     */
    @Override
    public void onBindViewHolder(@NonNull ViewHolder viewHolder, int i) {
        Song song = songCollection.get(i);
        viewHolder.songNameView.setText(song.getName());

    }
    /** Shows the number of items in the list
     * @return the number of values in the list
     */
    @Override
    public int getItemCount() {
        return songCollection.size();
    }
    /** Interface to set the onClick method in other classes*/

    public interface OnNoteListener{
        void onNoteClick(int position);
    }
}
```

Figure A.15: Complete SongAdapter class

Bibliography

[1] Mobile operating system market share worldwide - july 2019, 2019. http://gs.statcounter.com/os-market-share/mobile/worldwide.

[2] sklearn.preprocessing.maxabsscaler, 2019. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MaxAbsScaler.html.

[3] Richard G. Alonso M., David B. A study of tempo tracking algorithms from polyphonic music signals. 2003.

[4] Youssef M. Alzantot, M. Ubiquitous pedestrian tracking using mobile phones. 2012.

[5] AndroidDevelopers. Keeping your app responsive, 2019. https://developer.android.com/training/articles/perf-anr.

[6] AndroidDevelopers. Sensorevent, 2019. https://developer.android.com/reference/android/hardware/SensorEvent.

[7] AndroidDevelopers. Threadpoolexecutor, 2019. https://developer.android.com/reference/java/util/concurrent/ThreadPoolExecutor.html.

[8] Karageorghis C.I. Bacon C. J., Myers T. R. Effect of music-movement synchrony on exercise oxygen consumption. 2012.

[9] Noury N. Barralon P., Vuillerme N. Walk detection with a kinematic sensor: Frequency and wavelet comparison. 2006.

[10] Bailey B. P. Biehl J. B., Adamczyk P. D. Djogger: A mobile dynamic music device. 2006.

[11] Schedl M. Bock S. Enhanced beat tracking with context-aware neural networks. 2011.

[12] Widmer G. Bock S., Krebs F. Joint beat and downbeat tracking with recurrent neural networks. 2016.

[13] Harle R. Brajdic A. Walk detection and step counting on unconstrained smartphones. 2013.

[14] Zhang Y. Z. Chen G .L., Fei L. I. Pedometer method based on adaptive peak detection algorithm. 2015.

[15] Auld D. Multiple sensors on an android application, 2017. https://stackoverflow.com/questions/41859899/multiple-sensors-on-an-android-application.

[16] Schrauwen B. Dieleman S., Braken P. Audio-based music classification with a pretrained convolutional network. 2011.

[17] Aksoy S. Dirican A .C. Step counting using smartphone accelerometer and fast fourier trransform. 2017.

[18] Koppe E. Edel M. An advanced method for pedestrian dead reckoning using blstm-rnns. 2015.

[19] Daniel P. W. Ellis. Beat tracking by dynamic programming. 2007. https://librosa.github.io/librosa/generated/librosa.beat.beat_track.html?highlight=beat%20track.

[20] Dooley J. F. *Software Development, Design and Coding.* 1990.

[21] Ferris J. Let's get physical: The psychology of effective workout music, 2013. https://www.scientificamerican.com/article/psychology-workout-music/.

[22] Laroche J. Efficient tempo and beat tracking in audio recordings. 2003.

[23] Liu D. Jia B., Lv J. Deep learning-based automatic downbeat tracking: a brief review. 2019.

[24] Qi G. Kang X., Huang B. A novel walking detection and step counting algorithm using unconstrained smartphones. 2018.

[25] Priest D. L. Sasso T. A. Morrish D. J. Walley C. J. Karageorghis C. I., Mouzourides D. A. Psychophysical and ergogenic effects of synchronous music during treadmill walking. 2009.

[26] Karumi. Dexter, 2015. `https://github.com/Karumi/Dexter`.

[27] Jacobsen T. Schubotz R. I. Kornysheva K., Von Cramon D. Y. Tuning-in to the beat: Aesthetic appreciation of musical rhythms correlates with a premotor activity boost. 2010.

[28] H. Leppakoski. Error analysis of step length estimation in pedestrian dead reckoning. pages 1136–1142, 2002.

[29] Goto M. An audio-based real-time beat tracking system for music with or without drum-sounds. 2001.

[30] Sabatini A. M. Mannini A. A hidden markov model-based technique for gait segmentation using a foot-mounted gyroscope. 2011.

[31] Davies M. E. P. Klapuri A. McKinney M. F., Moelants D. Evaluation of audio beat tracking and music tempo extraction algorithms. 2007.

[32] Community Manager meahtenoha. Retirement of our running feature, 2018. `https://community.spotify.com/t5/Content-Questions/Retirement-of-our-Running-Feature/td-p/4383603`.

[33] Community Manager meahtenoha, 2019. `https://www.music-ir.org/mirex/wiki/MIREX_HOME`.

[34] mostar. Get multiple sensor data at the same time in android, 2012. https://stackoverflow.com/questions/12326429/get-multiple-sensor-data-at-the-same-time-in-android.

[35] Wirth N. Program development by stepwise refinement. 1971.

[36] Li Q. Asare P. Stankovic J. A. Hong D. Zhang B. Jiang X. Shen G. Zhao F. Nirjon S., Dickerson R. F. Musicalheart: A hearty way of listening to music. 2012.

[37] Flores-Mangas F. Oliver N. Mptrain: A mobile, music and physiology-based personal trainer. 2006.

[38] Keller T. Dietz V. Morari M. Pappas I. P. I., Popovic M. R. A reliable gait phase detection system. 2001.

[39] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[40] V. Garaj F. Cecelja W. Balachandran R. Jirawimut, P. Ptasinski. A method for dead reckoning parameter correction in pedestrian navigation system. 2001.

[41] Padmanabhan V. N. Sen R. Z. Rai A., Chintalapudi K. K. Zero-effort crowdsourcing for indoor localization. 2012.

[42] Bock S. Schluter J. Musical onset detection with convolutional neural networks. 2013.

[43] Chen Y. Chen H. H. Wang J. H., Ding J. J. Real time accelerometer-based gait recognition using adaptive windowed wavelet transforms. 2012.

[44] Edwards B. Waterhouse J., Hudson P. Effects of music tempo upon submaximal cycling performance. 2010.

[45] Huang B. Yang, X. An accurate step detection algorithm using unconstrained smartphones. 2015.

[46] Schnitzer A. Leonhardt S. Schiek M. Ying H., Silex C. Automatic step detection in the accelerometer signal. 2007.

[47] Chen Z. An lstm recurrent network for step counting. 2018.