SAPIENZA
UNIVERSITÀ DI ROMA

MSc in Data Science

FACULTY OF ENGINEERING IN COMPUTER SCIENCE AND STATISTICS

To fall or not to fall

STATISTICAL METHODS FOR MACHINE LEARNING

Professors:
Pierpaolo Brutti

Students:
Enrico Grimaldi
Nicola Grieco
Giuseppe Di Poce
Davide Vigneri

Academic Year 2022/23

Contents

1 Introduction

Falls are a critical concern in the realm of public health, particularly for the elderly population, where approximately 30 percent of those over 65 years living in the community experience a fall each year. The development and validation of fall detection algorithms have been hampered by the limited availability of real-world fall data.

In this challenging context, our efforts will encompass a comprehensive start-to-end pipeline that spans various stages, ranging from data collection to machine learning predictions.

Our goals will then be to:

- simulate a collection of falls by using specific sensors and experimenting with different fall modes
- try to mitigate the effect of bias due to the synthetic (simulation) approach by special feature extraction algorithms using a small amount of real falls data.
- develop fall classification algorithms both based on classical machine learning models and neural network models
- use the best model obtained to implement a practical solution to fall detection, based on the TinyML framework.

(a) Man before the fall (b) Man after the fall

Figure 1: Example of TinyML implementation of real-time fall detection and activation of a protection mechanism

2 Data collection

We have assembled a data set of simulated falls while using the Arduino model Nano 33 BLE Sense (Fig. 1), equipped with the necessary sensors. To realistically simulate falls, we have developed a compact torso that can be securely fastened to the participant's back during data collection. This unique approach has allowed us to capture a wide range of scenarios in various indoor and outdoor settings, ensuring the data set's authenticity.

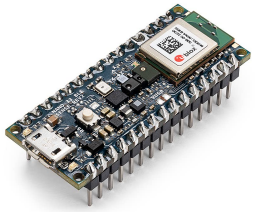


Figure 2: Arduino Nano 33 BLE Sense

The resulting data set comprises two primary signals: accelerometer and gyroscope, each possessing three axes (x, y, z) resulting in multidimensional timeseries records (Fig. 2). The sampling rate for both sensors is consistently set at 100 values per second, with data collected at regular 20-second intervals. However, to simplify the data set and make it more manageable for analysis, we have opted to retain and store only 400 timestamps for each observation (experiment). This reduction results in an effective sampling rate of approximately 20 values per second, which is around 20 percent of the sensors' default sampling frequency.

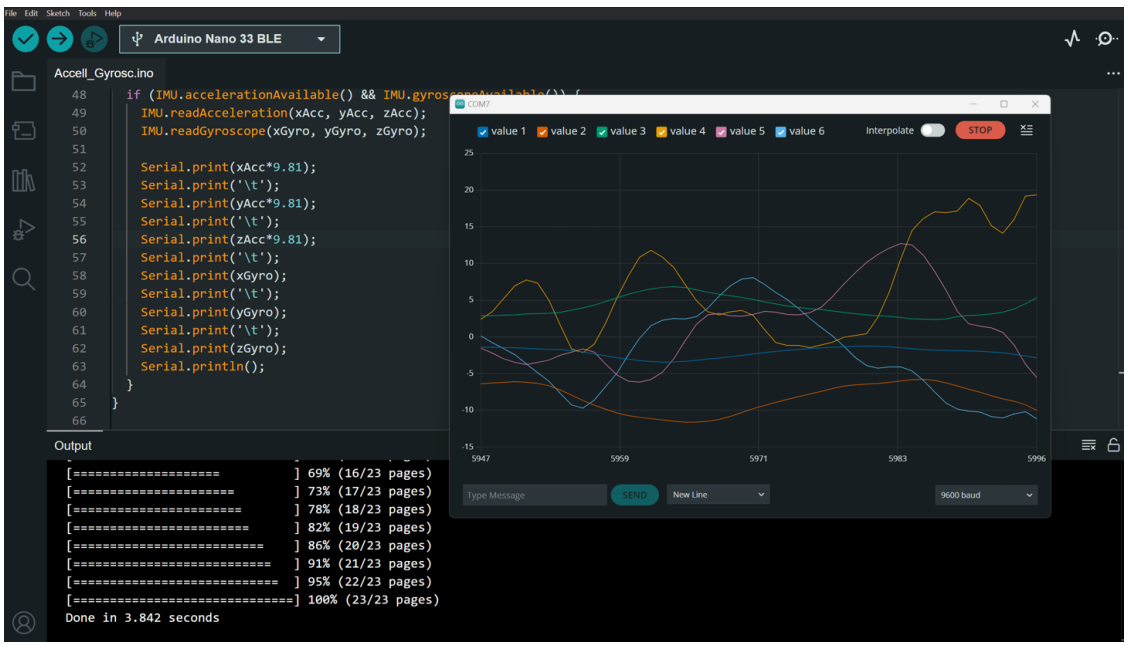


Figure 3: Serial plot of our six-dimensional data during collection.

Subsequently, the collected data was meticulously classified and labeled into seven distinct classes: walking, sitting, stepping, lateral falls, reverse falls, light falls, and severe falls. This diverse labeling allows us to conduct in-depth analysis and effective categorization of various activities and falls, providing valuable insights into the different scenarios that can lead to falls in real-life situations.

(a) Fall (b) Reverse fall (c) Step

Figure 4: Examples of collected classes: (A) fall, (B) reverse fall, (C) step

The resulting data set is stored in .CSV format, as illustrated in Figure 1. It's worth noting that this data set, while rich in information, may initially appear somewhat intricate and less intuitive due to its structure. In this data set, a single observation spans multiple rows, contrary to the conventional representation where a single observation is typically depicted in a single row.

3 Preprocessing and features extraction

In the preprocessing phase of our project, we meticulously prepared and transformed the collected data to ensure its suitability for further analysis. This critical stage involved two key steps aimed at enhancing the dataset's quality. Let's delve into the details of our preprocessing pipeline.

3.1 Data Flattening

The preprocessing phase opens up by "flattening" the dataset. Let's remind that each recording contained six signals representing accelerometer and gyroscope data across three axes (x, y, z). Nonetheless, the initial dataset that we obtained is not desirable since each column of stores all the data that refer to a particular signal. In other words, in the column "xAcc" of Figure 1 one could find all the time series referring to the acceleration on the x-axis. However, we know that a single observation is made of 6 time-series (one for each signal) each consisting of 400 entries (recall that we stored 400 discrete points of the continuous recording). Therefore, what we did was "flattening" the dataset by putting on the same row the six 400-long time series. As result, we obtain 242 rows and 2401 columns of which 2400 (400×2) are the features and 1 is the predictor variable. The result of the flattening process can be seen in Figure 2.

xAcc_1	xAcc_2	xAcc_3	xAcc_4	xAcc_5	xAcc_6	xAcc_7	xAcc_8	xAcc_9	xAcc_10
6.99	6.51	6.22	6.34	6.49	6.55	6.27	6.67	6.79	6.72
6.89	6.61	6.48	6.45	6.75	6.90	6.70	6.55	6.47	6.64
5.92	5.89	5.83	5.89	5.94	5.90	5.79	5.88	6.02	5.98
6.88	6.49	6.41	6.09	5.62	5.67	7.41	12.76	7.47	6.55
6.56	6.12	6.17	6.41	6.35	5.88	5.36	5.16	12.62	9.95
...	...	...	...	...	...	...	...	...	...
6.59	6.94	7.21	6.72	6.58	6.51	6.77	8.18	10.87	8.51
6.86	6.81	6.63	6.60	6.41	7.32	9.05	9.84	8.77	7.39
6.78	8.19	7.49	7.53	6.69	8.30	8.30	7.21	6.68	6.70
8.42	8.20	7.70	8.13	8.70	8.70	8.16	7.48	7.65	7.78
11.33	11.85	8.37	7.95	6.83	8.30	9.00	8.71	8.64	8.29

...	zGyro_392	zGyro_393	zGyro_394	zGyro_395	zGyro_396	zGyro_397	zGyro_398	zGyro_399	zGyro_400	label
...	-0.67	-1.10	-1.34	-1.28	-1.04	-0.98	-0.98	-0.98	-1.10	fall
...	-0.43	-0.73	-0.85	-1.04	-1.04	-1.34	-0.79	-0.92	-0.67	fall
...	22.77	23.86	26.43	34.36	39.92	39.67	37.78	38.09	36.19	fall
...	0.06	3.05	1.46	-3.05	-3.78	-4.94	-4.21	-2.56	1.40	fall
...	-16.78	-24.23	-29.91	-28.75	-20.26	-9.89	-9.58	-18.62	-35.58	fall
...	...	...	...	...	...	...	...	...	...	...
...	-6.47	-1.59	4.27	-4.70	-15.20	-17.40	-12.88	-16.66	-23.13	light
...	-14.71	-22.28	-26.18	-31.07	-37.66	-48.46	-50.66	-35.52	-21.06	light
...	16.24	18.68	18.80	18.31	14.28	5.86	4.33	15.14	24.60	light
...	-7.08	-10.86	-10.62	-5.62	1.71	11.35	15.69	18.62	22.95	light
...	-53.59	-46.45	-32.90	-32.04	-35.77	-34.55	-28.02	-22.52	-19.23	light

Figure 5: Data set after flattening

3.2 Computing the Norm for Acceleration and Gyroscope

In the process of extracting predictive variables, a crucial step involves computing the norm for both acceleration and gyroscope (angular velocity) signals. This computation serves to quantify the intensity of these signals in three-dimensional space for both the gyroscope and the accelerometer according to the definition of vectorial sum:

$$Acc = \sqrt{Acc_x^2 + Acc_y^2 + Acc_z^2}$$
$$Gyr = \sqrt{Gyr_x^2 + Gyr_y^2 + Gyr_z^2}$$

This approach is driven by the need to detect the force resulting from falls independently from the specific orientations of the wearable device. This way, when any axis exhibits a negative value, the power notation ensures it is transformed into a positive value.

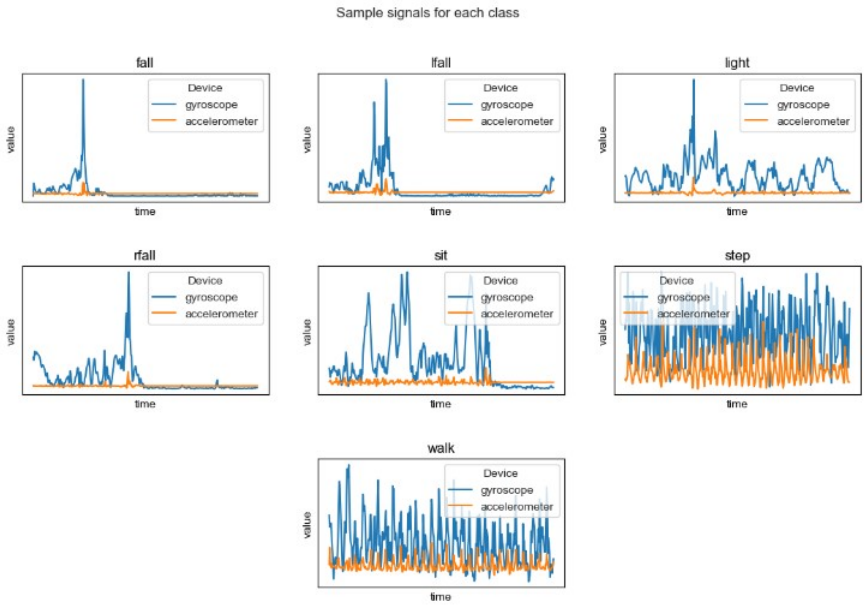


Figure 6: Examples of the resulting 1-dimensional signals

Then we can proceed with building an embedding that will transform our time series inputs, from the Arduino's gyroscope and accelerometer, into informative features in order to classify our falls.

3.3 Fast Fourier Transform

We decide to map the signal to the frequency domain using the Fourier Transform for two main reasons:

- to reduce the level of noise;
- to derive an extremely reduced number of features (it would be impossible to consider the 400 discrete values of each time series as features to feed to the machine learning model).

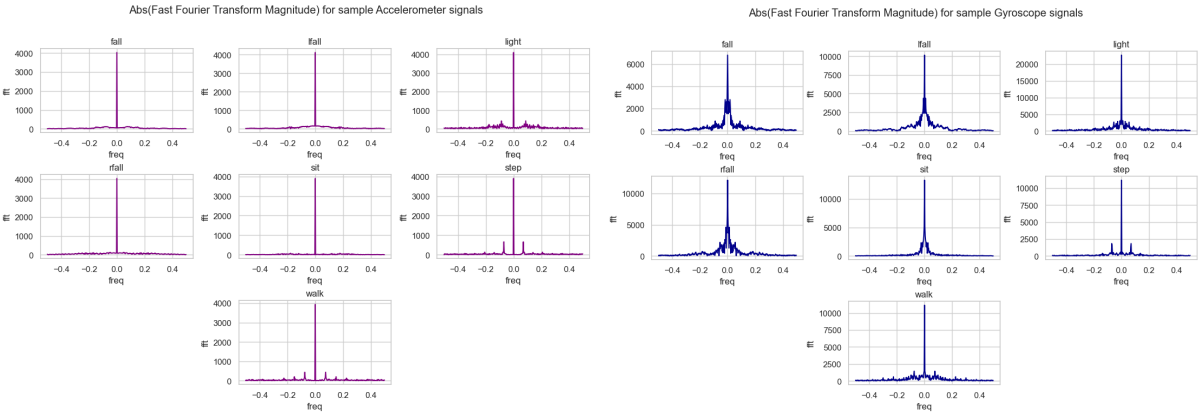


Figure 7: Fast Fourier Transform of sampled signals

We then used an FFT algorithm from the Scipy library (Fig. m) to transform each signal, from the output of the transformation we then proceeded with the following steps:

1. we consider only positive frequencies since the output is symmetrical with respect to zero;
2. we divide the power spectrum density into bins of equal size;
3. we take the peaks in terms of magnitude for each bin;
4. we treat only the peaks for the bins closest to frequency 0 as they have been empirically shown to be more significant.

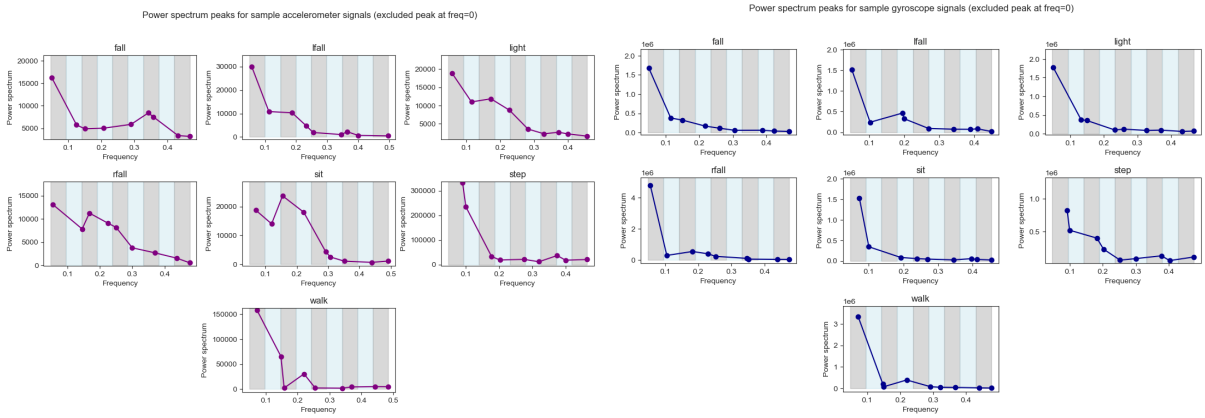


Figure 8: Dividing in bins and taking the peaks.

Finally, we decided to derive three different statistics from the power spectrum density that are considered informative in the literature:

- the median;
- the MAD (Mean Absolute Deviation);
- the skewness.

3.4 Mother wavelet comparison

At this stage of feature extraction, we were inspired by an article that identified an interesting feature for fall classification by thresholding algorithm. For the calculation of such a metric (which we will call "cwt coefficient") we will proceed as follows:

- from an online data set of real falls (not simulated by us) we extract the profile of signal peaks (of accelerometers) in a two-second window;
- from the above peaks we derive an average peak (an average signal lasting two seconds) that will represent our fall benchmark;

- we approximate this last average discrete signal to a "continuous" curve (specifically to a *Morlet wavelet*) by constructing a "mother wavelet."
- finally we cycle over each of our signals, identify a peak for that signal, and compare the trend of that peak to our mother wavelet via a similarity measure

Given $SV(t)$ the digital signal corresponding to the two-second window of a "candidate peak" detected by running along a timeseries of the accelerometer sum signal. Given Φ_{fall} the mother wavelet as defined above. The following coefficient estimate the similarity of the candidate window with the mother wavelet and with scaled and translated versions of it:

$$CWTcoeff(a,b) = \frac{1}{\sqrt{a}} \int_{-\infty}^{+\infty} SV_{candidate}(t) \cdot \Phi_{fall}\left(\frac{t-b}{a}\right) dt$$

The next flowchart defines the algorithm for extracting this new feature from the digital signals of the accelerometer only (the sum signal).

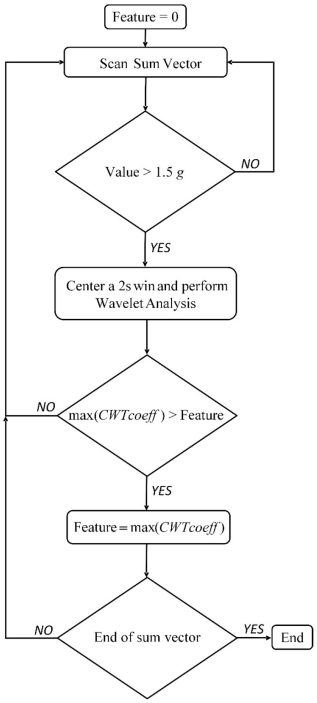


Figure 9: Algorithm for extraction mother wavelet based feature

The final data set will consist of the following features:

- 5 peaks of the magnitude of the power spectrum density for the accelerometer signal and 5 peaks for the gyroscope;
- 3 statistics (median, mad and skewness) for the two PSDs;
- 1 cwt coefficient for the similarity with the accelerometer peak mother wavelet.

4 Machine Learning Algorithms

Having finished the feature extraction part, we then try to build the final model that can best solve our classification problem. As we will see, further refinements will be needed to be applied to the data and fed into the pipeline.

4.1 Scaling

The extracted features are all of different scales and refer to rather different physical quantities or mathematical measures. We therefore find it necessary to scale the input data to all machine learning and deep learning models that will follow through stardadization.

4.2 Binary Classification

To perform the binary classification we have to do the reclassification of our dataset, which initially comprised seven distinct levels, into two main groups: "fall" and "normal". In our binary classification task, we initiated our exploration with fundamental models, such as Naive Bayes and Logistic Regression, as the initial benchmarks. While these models serve as essential starting points, we quickly transitioned to more advanced classification methods to harness the full potential of our data. To fine-tune and optimize our models, we employed the RandomizedSearchCV technique, which enabled us to systematically cross-validate various hyperparameters, ensuring that our models were tailored to the unique characteristics of our dataset. Evaluating the model performance on the test set was a crucial step in our analysis. This allowed us to objectively assess the effectiveness of our classification algorithms in real-world scenarios. To provide a visual representation of our classifiers' behavior, we leveraged Principal Component Analysis (PCA) to reduce the dimensionality of our data and plotted the decision boundaries of our models on the first two principal components.

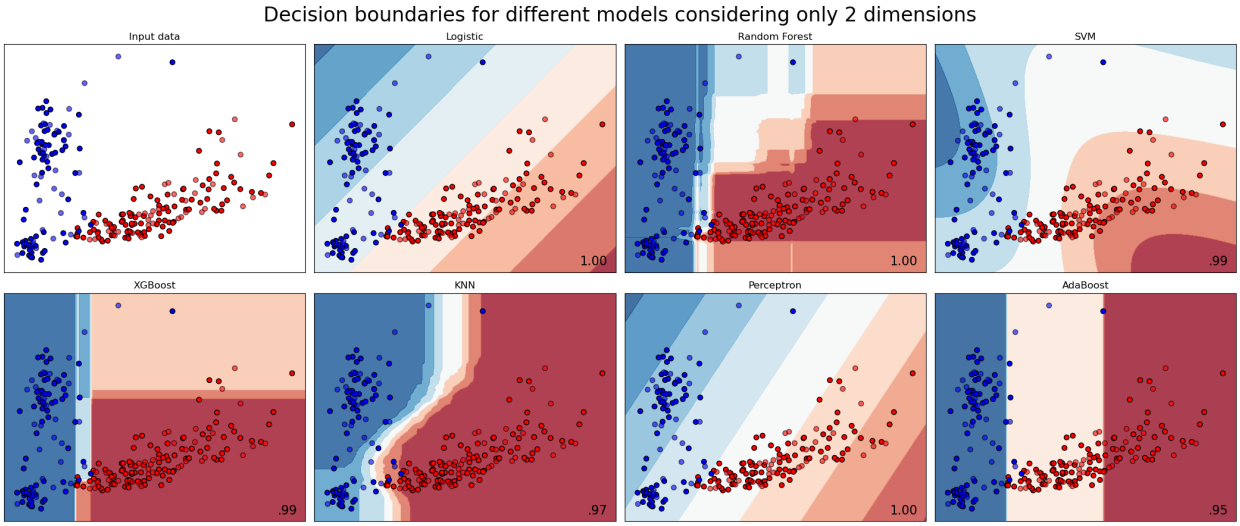


Figure 10: Decision boundaries in 2 dimensions

This two-dimensional visualization offered valuable insights into how our classifiers made

decisions and separated the binary classes. By juxtaposing the decision boundaries, we could observe how different models interpreted the data and discern their strengths and limitations in a concise and interpretable manner.

4.2.1 Performance Metrics

In this section, we summarize the outcomes of our classification experiments by listing the classifiers used and the key evaluation metrics employed to assess their performance. This overview serves as a concise reference for the subsequent detailed comparison of each classifier's performance in the table.

Classifier	Recall	Precision	F1-score	Accuracy	AUC
Logistic	0.974	0.973	0.973	0.973	0.974
Random Forest	0.987	0.986	0.986	0.986	0.987
SVM	0.960	0.961	0.959	0.959	0.960
ADAboost	0.973	0.973	0.973	0.973	0.973
KNN	0.961	0.961	0.959	0.959	0.961

Below, we present the ROC (Receiver Operating Characteristic) curves for all classifiers, providing a comprehensive visual representation of their respective performance in distinguishing between positive and negative instances

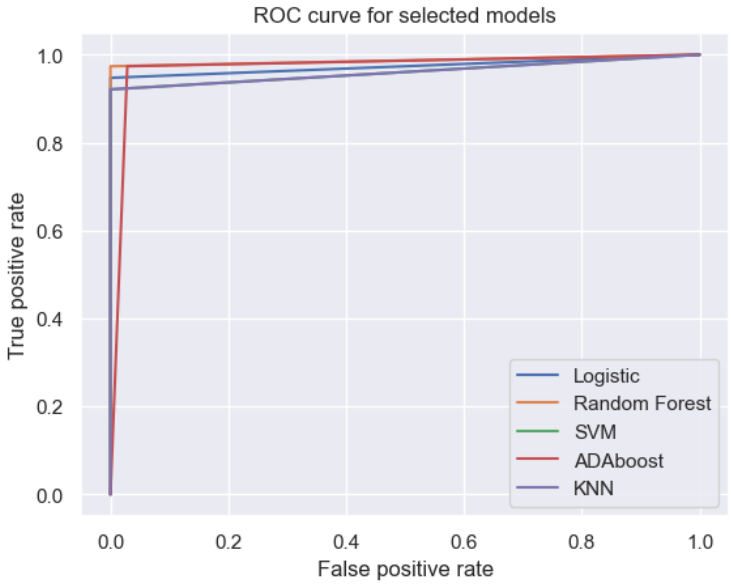


Figure 11: ROC curves for Binary Classifiers

Analyzing the ROC curves for all classifiers, we observe that most of them appear to extend to the upper-left corner of the graph. This indicates that these models are capable of achieving high true positive rates while minimizing false positives. However, it is important

to note that performance may vary depending on the application context. The presence of ROC curves extended towards the upper-left corner suggests good discriminative ability and increased sensitivity to positive instances, which is generally positive. This is particularly favorable in our context, as the positive class represents "falls," and our primary interest lies in accurately recognizing this class.

We employed the ROC curve to enhance our assessment but ultimately relied on Calibration Curves for the final decision. Calibration curves, also known as reliability curves, juxtapose actual or empirical probabilities against estimated or predicted probabilities. The calibration intercept, serving as an indicator of calibration-in-the-large, ideally has a value of zero. Values below the curve suggest overestimation, while values above the curve suggest underestimation.

Through this last visualization, AdaBoost emerged as the clear victor in our analysis. Our approach enabled us to assess and select the most effective classifier, providing valuable insights into the dataset's dynamics and the performance of various machine learning models.

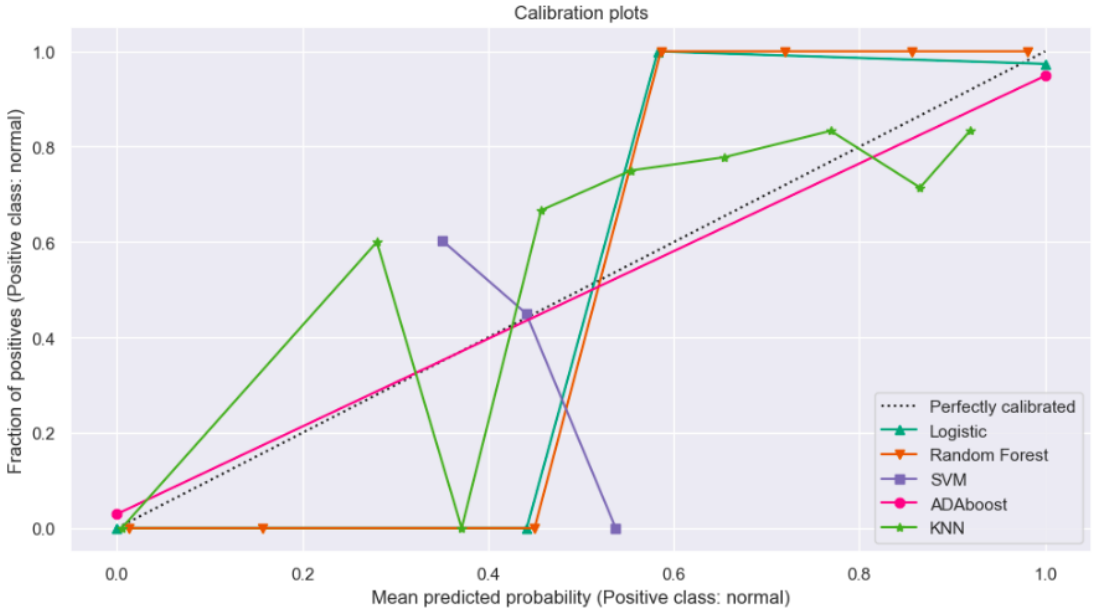


Figure 12: Calibration plots

4.3 Multi-class Classification

Stochastic Gradient Descent (SGD) is a simple yet very efficient approach to fitting linear classifiers and regressors under convex loss functions such as (linear) Support Vector Machines and Logistic Regression. It supports multi-class classification by combining multiple binary classifiers in a "one versus all" (OVA) scheme. For each of the classes, a binary classifier is learned that discriminates between that and all other classes. The figure below illustrates the OVA approach on the on our dataset considering only 2 features to visualize a 2-d plot. The dashed lines represent the seven OVA classifiers; the background colors show the decision surface induced by the seven classifiers. We used deafult options for the classifiers, only to show how it works.

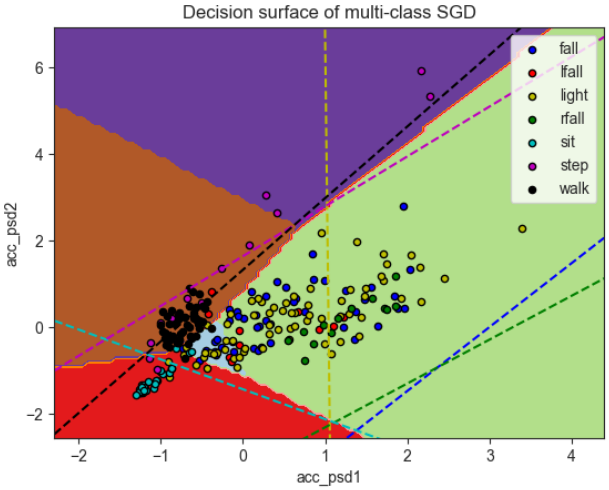


Figure 13: Multi-class surface in two dimensions

Furthermore we noticed that a probable issue could be the fact that we have a little number of samples for classes that represent the different types of fall.

4.3.1 Linear multi-classifiers by *SGD*

Basically what we are doing in this section is minimising the various cost functions using various linear classifiers as the chosen loss function varies. The following is a summary table of the results and a brief explanation of the chosen cost functions;

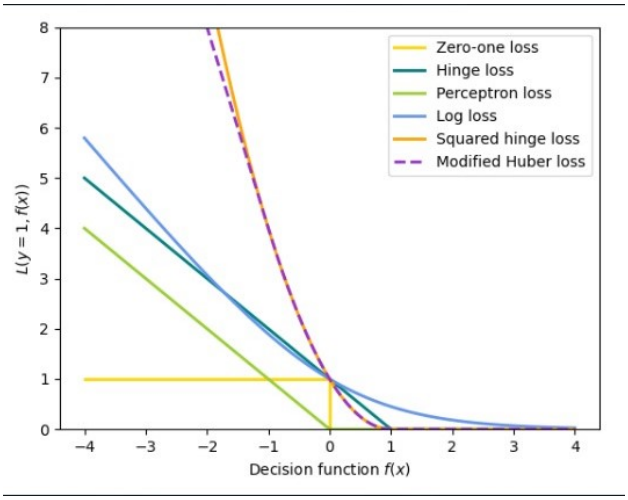


Figure 14: Objective function used and their shape

Losses formula:

- Logistic Regression → Log Loss = L(y$_i$, $f(x_i)$) = $log(1 + exp(-y_i f(x_i)))$
- Hinge Loss →L(y$_i$, $f(x_i)$) = $max(0, 1 - y_i f(x_i))$
- Squared Hinge Loss →[L(y, f(x)) = $\max(0, 1 - y \cdot f(x))^2$]
- Perceptron →L(y$_i$, $f(x_i)$) = $max(0, -y_i f(x_i))$

Objective function	Classifier	Balanced Accuracy	π
LogLoss	Logistic Regression	0.74	α=0.01;$L2$
Hinge Loss	SVM	0.80	α=0.01;$L1$
Squared Hinge	SVM	0.75	α=0.0001;$L1$
Perceptron	Perceptron	0.76	α=0.0001,$L1$
Modified Huber	SVM(γ=2)	0.68	α:0.01,$L1$

4.3.2 Linear and Quadratic Discriminant Analysis

Both LDA and QDA can be derived from simple probabilistic models which model the class conditional distribution of the data $P(X|y = k)$ for each class k.
For the purpose of our analysis we have tried using both classifiers with the following results:

Balanced Accuracy: 0.34
Classification Report:

	precision	recall	f1-score	support
fall	0.59	0.83	0.69	12
lfall	0.00	0.00	0.00	2
light	0.56	0.91	0.69	22
rfall	0.00	0.00	0.00	3
sit	0.00	0.00	0.00	8
step	0.00	0.00	0.00	2
walk	1.00	0.67	0.80	12
accuracy			0.62	61
macro avg	0.31	0.34	0.31	61
weighted avg	0.51	0.62	0.54	61

(a) QDA

Balanced Accuracy: 0.72
Classification Report:

	precision	recall	f1-score	support
fall	0.75	0.75	0.75	12
lfall	0.00	0.00	0.00	2
light	1.00	0.95	0.98	22
rfall	0.33	0.33	0.33	3
sit	1.00	1.00	1.00	8
step	1.00	1.00	1.00	2
walk	1.00	1.00	1.00	12
accuracy			0.87	61
macro avg	0.73	0.72	0.72	61
weighted avg	0.89	0.87	0.88	61

(b) LDA

Figure 15: QDA and LDA performances

It's evident how quadratics performs really badly differently from the Linear. In particular the *Linear* one can be considered as a special case of QDA, where the Gaussians for each class are assumed to share the same covariance matrix: $\Sigma_k = \Sigma$ for all k.

The log-posterior of LDA can also be written as:

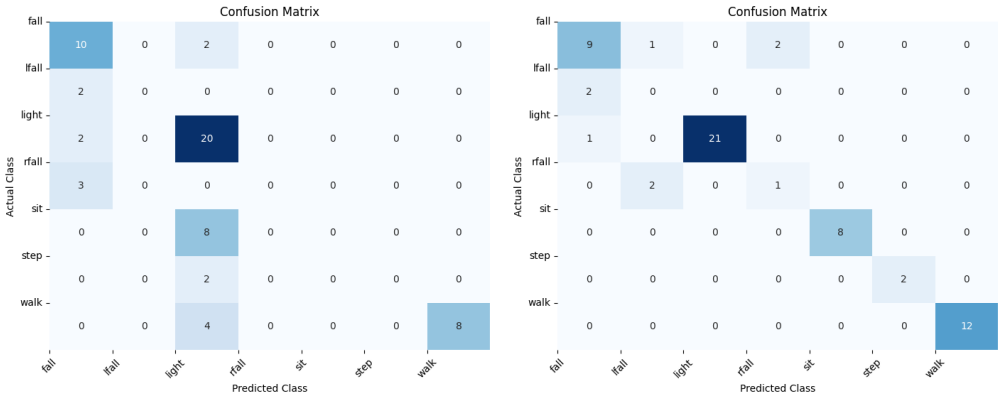
$$logP(y = k|x) = \omega_k^t x + \omega_{k0} + Cst$$

where $\omega_k = \Sigma^{-1}\mu_k$ and $\omega_{k0} = -\frac{1}{2}\mu_k^t\Sigma^{-1}\mu_k + logP(y = k)$

These quantities correspond to the coefficient and intercept attributes, respectively. From the above formula, it is clear that LDA has a linear decision surface.
It' simpprtant to fix tha LDA assumption:
Assumption (A.lda):

in each class k the joint distribution of the features is a multivariate normal centred on the mean vector μ_k , and having a covariance matrix Σ common to all classes, i.e.

$$X|Y = k \sim N(\mu_k; \Sigma) \text{ for each k} = 1, 2, ..., K$$



(a) QDA (b) LDA

Figure 16: QDA and LDA confusion matrices

Implications:

- class conditional densities are $f_k(x) = (x; \mu_k; \Sigma)$
- the scatter of the points in each class have elliptic-symmetric form
- since the features have covariance matrix in each class, the assumption (A.lda) implies that in all classes the scatter of the features has homogeneous geometric characteristics. LDA is also a 'linear classification method' because it produces linear decision boundaries. It can be shown that the LDA produces decision boundaries between classes that consist of straight lines (if p = 2), planes (p = 3), hyper-planes (when $p > 3$, as in our case).

4.4 Resampling

We note that some metrics remain very close to zero in the multi-label case, and this is probably due to the absence of examples of some classes in the training set. As we can see from the comparison we have an unbalanced data set for some classes and this could lead to problems in classification: we do not train the model on enough examples for each label and it is quite difficult to distinguish between classes belonging to the same macro-class (i.e. fall and normal).

We then use a resampling technique called SMOTE, in particular we have selected a variant that performs over and undersampling based on clustering results. In this way we try to ensure a data set that is as balanced as possible.

KMeans SMOTE uses a KMeans clustering method before to apply SMOTE. The clustering will group samples together and generate new samples depending of the cluster density.

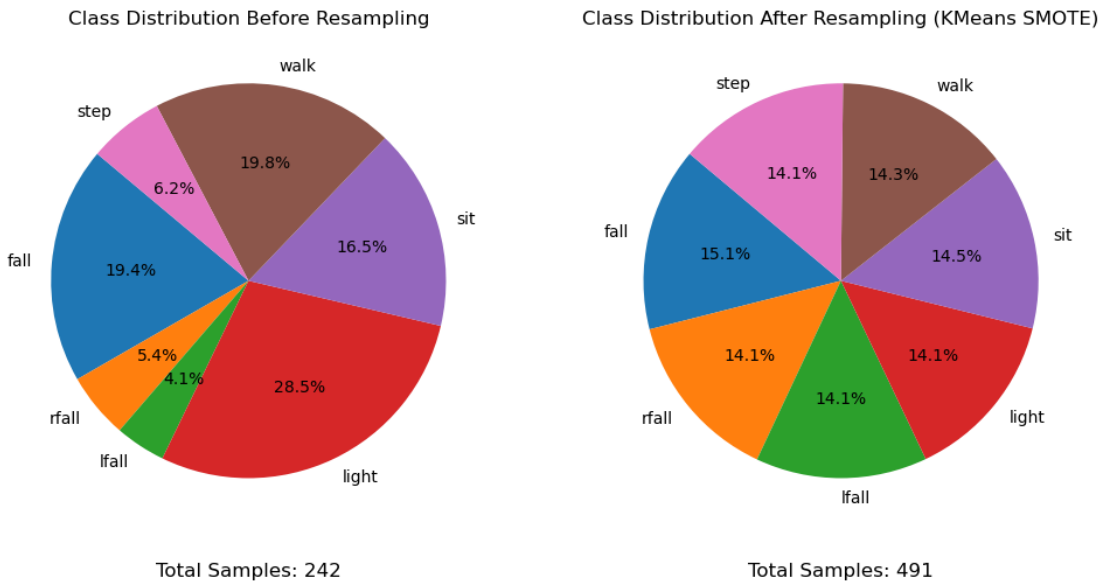


Figure 17: Class unbalanced distribution vs class distribution after resampling

We then verify an improvement in performance when we resample and using a Softmax model.

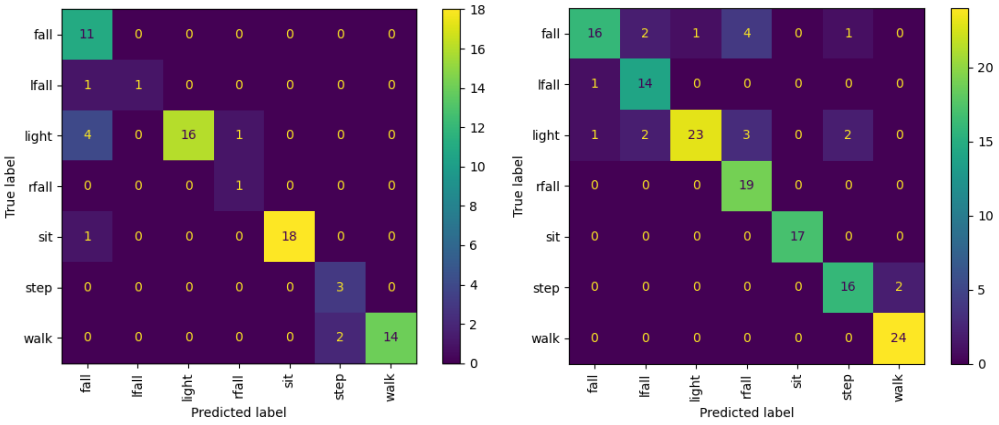
	precision	recall	f1-score	support
fall	0.65	1.00	0.79	11
lfall	1.00	0.50	0.67	2
light	1.00	0.76	0.86	21
rfall	0.50	1.00	0.67	1
sit	1.00	0.95	0.97	19
step	0.60	1.00	0.75	3
walk	1.00	0.88	0.93	16
accuracy			0.88	73
macro avg	0.82	0.87	0.81	73
weighted avg	0.92	0.88	0.88	73

(a) Before

	precision	recall	f1-score	support
fall	0.89	0.67	0.76	24
lfall	0.78	0.93	0.85	15
light	0.96	0.74	0.84	31
rfall	0.73	1.00	0.84	19
sit	1.00	1.00	1.00	17
step	0.84	0.89	0.86	18
walk	0.92	1.00	0.96	24
accuracy			0.87	148
macro avg	0.87	0.89	0.87	148
weighted avg	0.88	0.87	0.87	148

(b) After

Figure 18: Performances before and after resampling using a Softmax classifier



(a) Before

(b) After

Figure 19: Confusion matrices before and after resampling using a Softmax classifier

Finally, however, we notice a problematic class both before and after applying the resampling technique: other labels including "lfall" and "fall" are often mistaken for "light." We need to better inspect this issue by analytically analyzing the level of "consistency" of the specific labels. We will use unsupervised learning techniques for this.

4.5 Unsupervised Learning

In this section of our project we will investigate about the similarities among our data collected using unsupervised methods: the main idea is to highlight 'overlapping' among different labels in order to be able to collapse some classes, gaining a deeper understanding of the underlying structure in the data. This is due to the fact that after a first algorithmic analysis we find out that some labels seems to be redundant and we want to catch overlaps between observations belonging to these classes that appear to be redundant.

4.5.1 Decomposition step

As first we want to "shrink" of our data: Using a dimensionality reduction like PCA basically we are reducing our data in order to be able to visualize it at maximum in three dimensional space. After a standard scaler of our data as $z = \frac{x_i - \mu}{\sigma}$ we are searching for the best trade off in terms of number of dimensions and explained variance of the components.

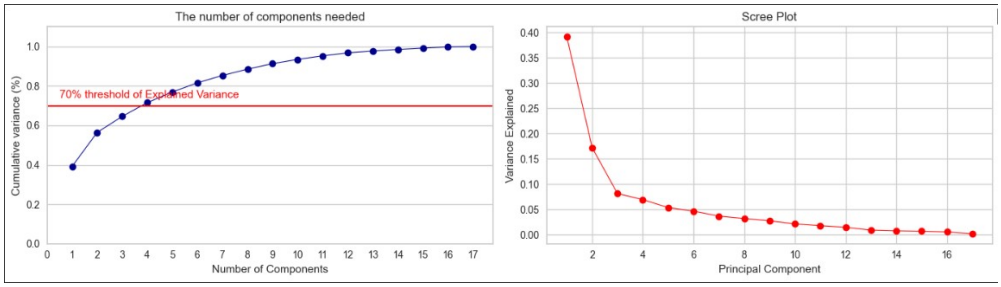


Figure 20: Cumulative Variance and scree plot of principal components from PCA;

As the above figure show, summing up the explained variance of the first three components we have the 65.0% of explained variance. For the aim of our analysis we chose to take only the first three components: please notice that this is not a theoretically correct choice since the components we are using explain little variance from what we would like. However, since the *scree plot* suggest an elbow with componets=3 and due to the fact that we want to visualise our data we will choose the first three components and perform an unsupervised analysis on them. The choice of being able to visualise data in a three-dimensional space will be very useful in identifying overlapping classes.

4.5.2 K-Means Algorithm

In order to groupify by similarities the data points we will implement a k-means++ algorithm. In order to choose the correct number of centroids by elbow method we will use YellowBrick, that is a powerful tool that allows us to use a silhouette coefficient as metric to compute the

intra-cluster variance (that is actually what we are more interested in), in order to choose the best k value. Specifically Silhouette Coefficient method is a tool for interpretation and validation of consistency within clusters of data, describing how good each object has been classified. The silhouette value is a measure of how similar an object is to its own cluster compared to other clusters, defined as:

$$S(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

where:

- $b(i)$ is the smallest average distance of point i to all points in any other cluster;
- $a(i)$ is the average distance of i from all other points in its cluster.

From the Silhouette Elbow method implemented as described we figure out an optimal $k = 5$. By the way we implemented the algorithm in a range of $k \in \{3, 6\}$ in order to control slight changes and clusters behaviour. A very good visualisation for our objective is the following histogram showing the class absolute frequencies of the observations in each cluster.

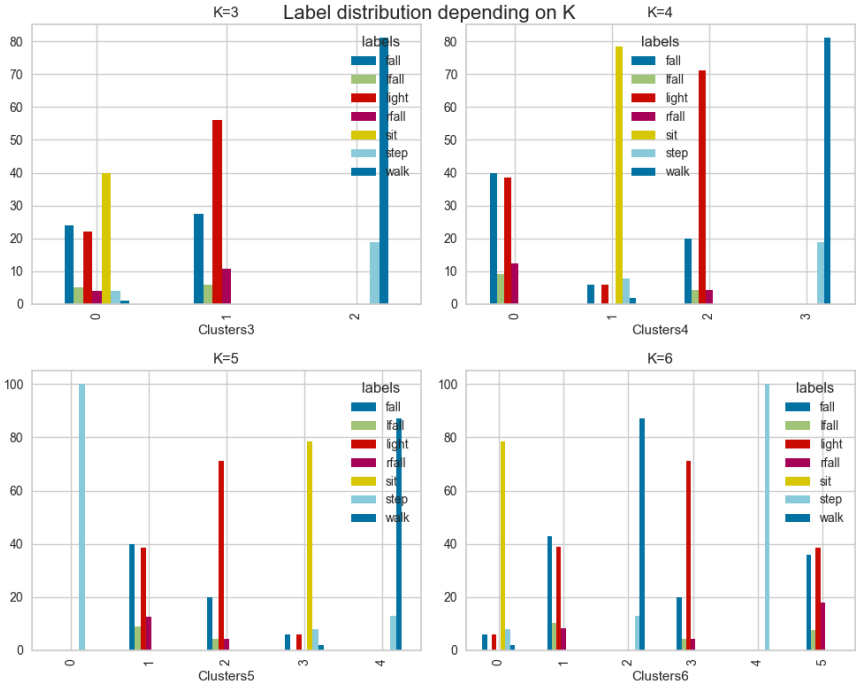


Figure 21: Absolute frequencies of labels depending on different K values;

We can clearly see, also from the distribution table below, how the clusters are variegated, in particular notice how, also increasing at $k = 6$ the number of centroids labels like $fall$ and $light$ overlaps in all the clusters.

distribution_6

labels Clusters6	fall	lfall	light	rfall	sit	step	walk
0	5.88	0.00	5.88	0.00	78.43	7.84	1.96
1	42.86	10.20	38.78	8.16	0.00	0.00	0.00
2	0.00	0.00	0.00	0.00	0.00	12.96	87.04
3	20.00	4.44	71.11	4.44	0.00	0.00	0.00
4	0.00	0.00	0.00	0.00	0.00	100.00	0.00
5	35.90	7.69	38.46	17.95	0.00	0.00	0.00

Figure 22: Table of percentage labels presence in each cluster

This is a symptom of a clear overlapping of the labels of these observations, which are too similar.
Inspired by true application of this kind of technology, we simply want to build a model that can recognise a fall with the highest possible success. Due to this we decide to collapse the overlapping labels into a single $fall$ one.

4.5.3 Calinski Harabasz score

Calinski-Harabasz method in unsupervised learning is also called *the variance ratio criterion (VRC)*. The idea is that well-defined clusters have a large between-cluster variance and a small within-cluster variance so the optimal number of clusters corresponds to the solution with the highest Calinski-Harabasz index value. This score attempts to measure how separate the clusters are from each other and how compact they are within each cluster. In general, a higher Calinski-Harabasz score indicates better clustering quality.

$$CH(k) = \frac{B(k)}{W(k)} \cdot \frac{n-k}{k-1}$$

where:

- B(k)= between cluster variation
- W(k)= within cluster variation
- n= number of data points
- k= number of clusters

By the Calinski test we may choose k=4. By the way this is just to figure out the best way to group our data by similarity and be able to visualise it, for the purpose of our supervised analysis we will try to be as conservative as possible by preserving the starting labels, collapsing only the initial 'light' into the 'fall'. At the end we end up with the following representation in $3d$ of our data points.

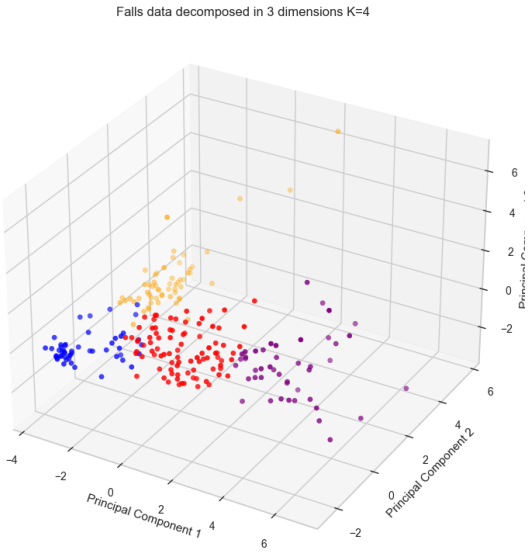
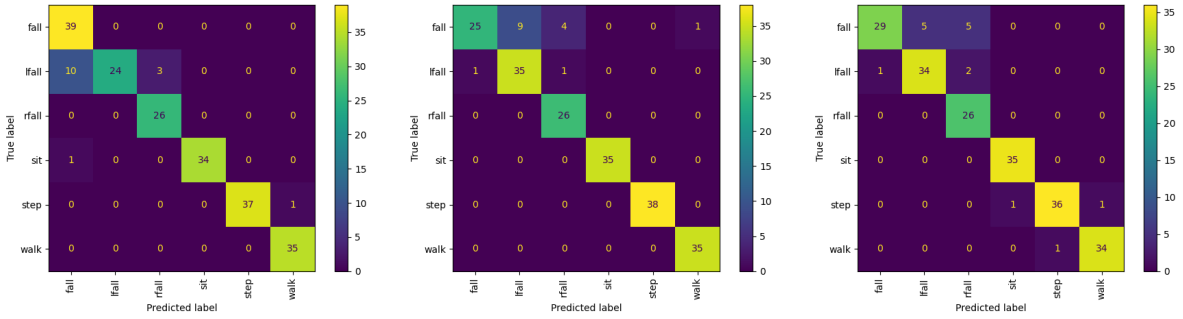


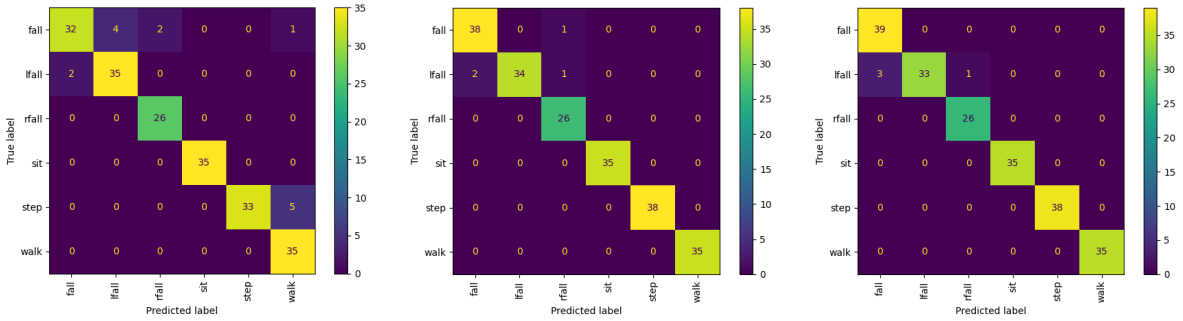
Figure 23: Final k-means grouping with k=4

4.6 Final multi-class models comparison

Basically in our first analysis with binary and multinomial classification we end up with the conclusion that some label were overlapping. We use unsupervised tools to investigate about it and we conclude that the *light* can be collapsed into the *fall* one. This is a choice made in order to be able to slightly improve our result and be able to distinguish in the best way our fall detection labels. Let's have a look to our final results after the reduction in the number of target labels:



AdaBoost | Softmax | K-NN



Perceptron | Svm | Random Forest

Table 1: Final multinomial classifier performances with six classes

We can clearly see that we end up with Random Forest that is a very strong classifier and seem to be the best one in our 'zoo classifier'. By the way notice that also AdaBoost perform very well and result to be the most calibrated one as we see in the binary classification case.

5 Deep Learning

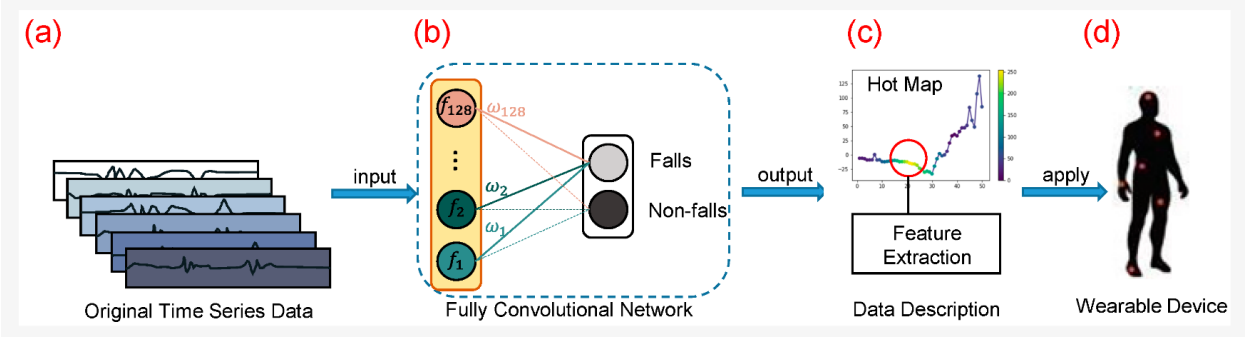


Figure 24: Convolutional Neural Network for fall detection

Neural networks excel in feature extraction due to their ability to automatically learn hierarchical representations from raw data, often surpassing human-engineered features. This capability arises from their capacity to capture complex, non-linear relationships within the data.Neural networks consist of multiple layers of interconnected artificial neurons, and as data flows through these layers, they adapt their internal parameters through training to identify discriminative patterns and features. Also in the context of fall classification using time series of acceleration and gyroscope data along the three dimensional axes, neural networks demonstrate a remarkable ability to autonomously discover intricate spatiotemporal patterns and features within the sensor data, surpassing the limitations of manually crafted feature engineering and offering improved accuracy in identifying various types of falls. We constructed several mostly convolutional neural networks and used different architectures; below is a list of the networks we used with the respective number of parameters and the accuracy evaluated on the validation set.

Architecture	Loss	Accuracy	Parameters
CNN	0.0587	0.9752	408679
CNN-HE	0.1058	0.9669	3193895
CNN-3B3Conv	0.1954	0.9504	1749127
CNN-EDU	0.0342	0.9917	407383
CBAM-IAM-BiLSTM	0.0959	0.9752	163159
CBAM-EDU-BiLSTM	0.0206	0.9835	228071

CNN = Convolutional Neural Network
CNN-HE = Convolutional Neural Network with Heuristic Enhancements
CNN-3B3Conv = Convolutional Neural Network with Three Blocks of Three Convolutions
CNN-EDU = Convolutional Neural Network with Educational Enhancements
CBAM-IAM-BiLSTM = Convolutional-Bidirectional LSTM with Attention Mechanism
CBAM-EDU-BiLSTM = Convolutional-Bidirectional LSTM with Educational Enhancements

The best model is the last one which is trained using the Adam optimizer with a learning

rate of 0.001. During training, the model is trained for 50 epochs with a batch size of 32. The performance of the model is evaluated on a validation dataset, and metrics such as accuracy are computed. This CBAM-EDU neural network effectively combines convolutional and bidirectional LSTM layers with educational enhancements, including the GELU activation function, to classify human activities and fall behaviors, capturing both spatial and temporal features from accelerometer and gyroscope data while addressing overfitting with dropout regularization. In the following plot is shows the metrics result in terms of accuracy and loss:

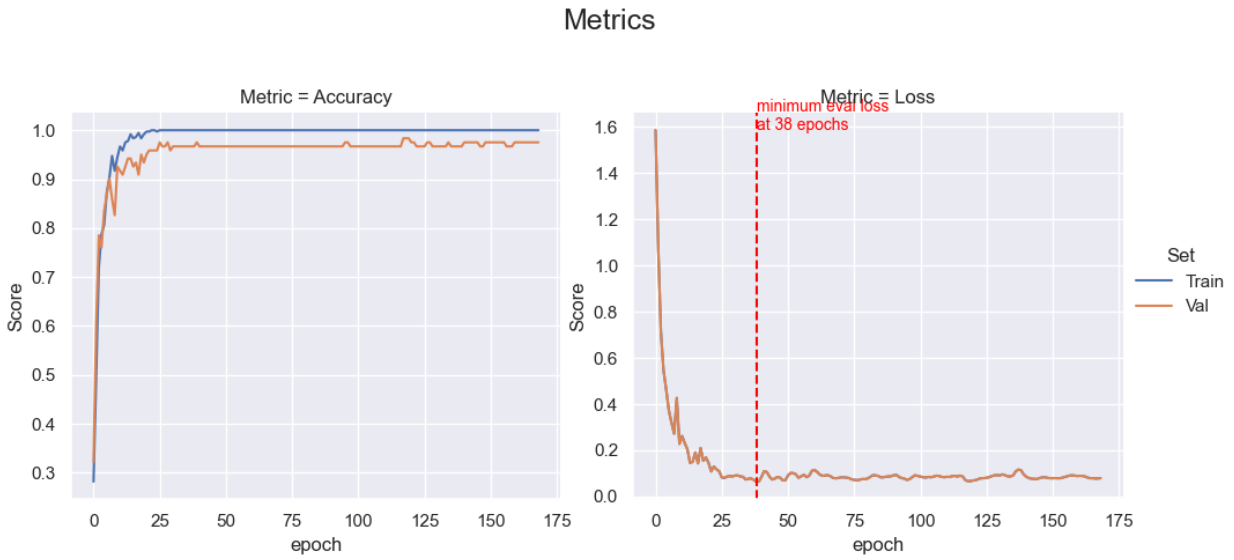


Figure 25: Accuracy and Categorical Crossentropy Loss using CBAM-EDU-BiLSTM

Here we can see that the loss reaches a minimum after 38 epochs adn then reaches a convergence.

5.1 Generate a compressed (TFLite) model

Once we had a well-performing deep learning model with TensorFlow core, the next challenge was deploying it on resource-constrained devices like Arduino. Model compression techniques were applied to reduce the model size while maintaining reasonable performance:

- Quantization: We converted the model's parameters to lower bit-width values.
- Pruning: Unimportant model parameters were pruned.
- Knowledge Distillation: We transferred knowledge from the complex model to a simpler one, suitable for Arduino.

We decide to apply the first technique, the quantization, so we can convert it to a smaller, more efficient ML model format called a TensorFlow Lite model. This process involves reducing the precision of the model's weights, and potentially the activations (output of each layer) as well. This reduction in precision leads to significant memory savings without

compromising on accuracy. Moreover, quantized models tend to execute faster due to the simpler calculations required. We performed the model conversion twice: once with quantization and once without. This allowed us to compare the results and choose the most suitable version for deployment.

5.2 Compressed vs original model performance

After compressing the model, we conducted a an evaluation to assess its performance on the test set. Remarkably, the model's accuracy remains consistent, showcasing no significant loss in its predictive capabilities. Prior to compression, the model achieved an accuracy of 0.9835 on the test set, and post-compression, the accuracy remained at an impressive 0.9835. This retention of accuracy underscores the effectiveness of the compression techniques employed, as they allowed us to streamline the model for deployment on resource-constrained devices without compromising its classification performance. This outcome reaffirms the feasibility of utilizing the compressed model on platforms like Arduino for real-time activity classification, while preserving the high level of accuracy achieved in the original deep learning model.

5.3 Generate a TensorFlow Lite for Microcontrollers Model

To enable seamless integration of our TensorFlow Lite quantized model with the Arduino IDE, we undertook the conversion process, transforming the model into a C source file that can be readily loaded by TensorFlow Lite for Microcontrollers. We first ensured that the necessary tool, xxd, was available for this purpose. Using a series of terminal commands, we initiated the conversion process by generating a C source file, "model.h," containing the model in a suitable format for Arduino. The code snippet below illustrates the sequence of commands used for this purpose:

```
# Install xxd if it is not available
!apt-get update && apt-get -qq install xxd
!echo "const unsigned char model[] = {" > /content/models/model.h
!cat models/model.tflite | xxd -i      >> /content/models/model.h
!echo "};"                              >> /content/models/model.h
```

Figure 26: code to convert TensorFlow Lite quantized model into a C source file

This resulting "model.h" file is a crucial component in the deployment process, as it allows the Arduino IDE to easily load and utilize the TensorFlow Lite quantized model, facilitating real-time action classification on the Arduino platform.

5.4 Microcontrollers and Arduino IDE

In order to enable real-time classification of human actions using our compressed TensorFlow Lite model on the Arduino platform, we implemented a series of critical components and

processes. Firstly, we incorporated the necessary libraries and tools, such as Arduino LSM9DS1, TensorFlow Lite for Microcontrollers, and specific TensorFlow Lite operations, into our Arduino sketch. This ensured seamless integration and compatibility between the hardware and software components of our system.

Secondly, we initialized the IMU (Inertial Measurement Unit) for data collection, allowing us to continuously receive accelerometer and gyroscope data in real time. This data serves as the input to our TensorFlow Lite model. To facilitate the integration of our model into the Arduino environment, we employed the "model.h" file, which contains the quantized model converted into a C source file format. This file was instrumental in enabling the Arduino IDE to load and execute the model on the Arduino board.

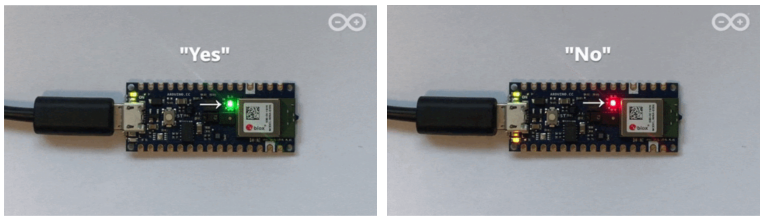
In addition to these key components, we implemented data preprocessing in real-time using the "StandardScaler." This scaler ensured that the incoming sensor data was standardized, aligning it with the preprocessing applied during model training. The real-time data preprocessing was critical in ensuring that the model received inputs in a consistent format, allowing for accurate and real-time classification of human actions.

With these components and processes in place, our Arduino-based system is capable of continuously collecting sensor data, preprocessing it, and running inference on the TensorFlow Lite model. The output from the model provides real-time classification of various human actions, making it a valuable tool for applications in health monitoring, security, and more, where instantaneous action recognition is crucial.

6 Further improvements

We have reached the culmination of our current project, successfully deploying our human action classification system on an Arduino Nano 33 BLE Sense. However, our journey doesn't end here. We have two exciting ideas for future enhancements

1. **Light Activation Based on Classes**: Implement a program on the Arduino to control the activation of lights or other external devices based on the detected classes of human actions. This feature allows the Arduino to autonomously trigger different responses, such as turning on lights when it classifies a "fall" or "light fall" action. It can be achieved by linking the output of the TensorFlow Lite model to Arduino's GPIO pins or suitable interfaces, enabling it to interact with the surrounding environment. Such automation can have applications in areas like home safety and healthcare, where the system can respond to detected actions in real-time.



(a) Fall (b) Not fall

Figure 27: Examples of collected classes: (A) Fall (B) Not fall

2. **Mobile Application for Results Visualization**: Develop a simple mobile application that can receive and display the real-time results of human action classification performed by the Arduino. Utilize the Bluetooth capabilities of the Arduino Nano 33 BLE Sense to enable communication between the Arduino and the mobile app. The application can provide a user-friendly interface for users to monitor and review the detected actions conveniently on their smartphones. This interface can offer real-time insights, historical data logging, and potentially additional features, making it an accessible and informative tool for various users, including caregivers, medical professionals, and individuals concerned about their safety.

Figure 28: Mobile app for fall detection