LIÈGE université
School of Engineering

Faculty of Applied Sciences

ELEN0062-1 Introduction to machine learning

Report - Project 3 - Human Activity Recognition

Professors :
GEURTS Pierre
WEHENKEL Louis

Report done by :
Florea Robert s201068
Guenfoudi Ihabe s194071
Ntakirutimana Renaux s202910

20 décembre 2024

Table des matières

1 Introduction

Human Activity Recognition is a key challenge in machine learning. Using time series measurements captured from a set of 31 sensors, this project focuses on predicting specific activities based on these data.

In this report, we describe the methodology adopted for data preprocessing, the analysis of different machine learning algorithms, and compare their performance and development.

2 Data preprocessing

2.1 Handling missing values

The first step in preprocessing was handling missing values.

For the learning set, we employed an conditional mean imputation approach, where missing values at a time step i were replaced with the mean of the same time step i across other recordings of the same activity.

For the test set, since the activity corresponding to each record is unknown (as determining it is the goal of the project),we used an unconditional mean imputation approach. Here missing values at a time step i were replaced with the mean of all activities at the same time step i.

These methods were chosen for their simplicity and computational efficiency, making it well-suited for our dataset's structure and size. Furthermore, for the learning set, as we use conditional mean imputation, it avoids the bias introduced by simple mean imputation.

While these approaches reduces variability in the data, this was not a significant concern as we choose not to address outliers in this project. This decision was made to maintain focus on handling missing data effectively without adding additional complexity to the preprocessing.

2.2 Computation of Metrics

Directly providing the raw data to the model would require a highly complex model to capture all the relevant patterns. To simplify the learning process and improve model performance, several metrics were computed to summarize key characteristics of the time series data.

These metrics are :

- *average* : The mean of the signal.
- *deviation* : The standard deviation of the signal.
- *minimum* : The minimum value of the signal.
- *maximum* : The maximum value of the signal.
- *median_ val* : The median value of the signal.
- *skew* : The skewness of the signal, which measures the asymmetry of the distribution (positive skew indicates a longer tail to the right, negative skew indicates a longer tail to the left).
- *kurt* : The kurtosis of the signal, which measures the sharpness of the peak of a frequency-distribution curve.
- *span* : The range of the signal, calculated as the difference between the maximum and minimum values.
- *variance* : The variance of the signal.
- *p25* : The 25th percentile of the signal.
- *p75* : The 75th percentile of the signal.
- *zero_ cross* : The normalized count of zero-crossings in the signal, representing the frequency of sign changes.
- *signal_ energy* : The mean energy of the signal, defined as the average of the squared amplitudes.
- *spec_ centroid* : The spectral centroid of the signal, representing the center of mass of the spectrum.
- *spec_ bandwidth* : The spectral bandwidth, measuring the dispersion of frequencies around the spectral centroid.

3 Methodology

To ensure a smooth and consistent workflow across different models, we established a general protocol, outlined below :

1. Load and process the data using the `load_and_prepare` function that prepares the data for the training using the principles described in the previous section.
2. Split the learning set into 80% training and 20% validation subsets using `train\_test\_split` from `fromsklearn.model_selection` (renamed as `split_data` in our code)
3. Use the `Pipeline` method from `sklearn.pipeline` (renamed `ML_Pipeline` in our code) to generalize the learning. This function is used such as :

   ```
   model_pipeline = ML_Pipeline([('transformation', Transformation()),('model', Model())])
   ```

   where 'transformation' is the name of the transformation that we want to apply to our data (followed by its corresponding method) and 'model' is the name of the model that we will use (once again, followed by its corresponding method). In our tests we standardized the data using `Scaler()` method from `sklearn.preprocessing` (original name of the method is `StandardScaler`).
4. Create a dictionary with the names (as keys) of the hyperparameters of our model and the values that we want to try for each one of them.
5. Find the best hyperparameters among the ones in the dictionary using `RandSearch` method from `sklearn.model_selection` (original name is `RandomizedSearchCV`). We will create a `RandSearchobject` with the following parameters :

   ```
   search = RandSearch(estimator=model_pipeline, param_distributions=parameters, n_iter=20,
                       scoring='accuracy', cv=5, n_jobs=-1, verbose=2, random_state=42)
   ```

   where :
   - *estimator* : is the machine learning model for which hyperparameters are being optimized
   - *param_ distribution* : the dictionary described in step 4.
   - *n_ iter* : the number of hyperparameter combinations to evaluate. Increasing this value improves the chances of finding better hyperparameters but also increases computation time.
   - *cv* : the number of cross-validation folds. Determines how the data is split for training and validation during the search.
   - *scoring* : the metric used to evaluate model performance. We based the search on the best accuracy.
   - *n_ jobs* : the number of parallel jobs to run. By setting it to -1 we use all available CPU cores.
   - *verbose* : controls the quantity of details of the output
   - *random_ state* : setting a state for reproducibility

   We will then apply `search.fit()` on our learning set and this is when the cross validation will be done.
6. Evaluate the model by calculating the accuracy, reviewing the best hyperparameters identified, analysing the most important features and other metrics.

4 Investigated approaches

In this section we will look at different machine learning algorithms and compare different metrics to choose the best one.

Of course, the evaluation of each model will be done using the protocol described in the previous section.

4.1 Random Forest

Random Forest is an ensemble technique (meaning it combines the predictions of multiple models to produce a more accurate and stable output) that computes multiple decision trees during training and aggregates their outputs. This method is very robust to overfitting and small changes in the data due to the "averaging" nature of the algorithm. Therefore, Random Forest reduces variance and improves generalization.

Let's look at the impact of the hyperparameters on the model's accuracy. First we define the domain from which our cross-validation will try the hyperparameters :

```
parameters = {'rf_step__n_estimators':[100, 200, 300], 'rf_step__max_depth':[10, 20, 30, None],
              'rf_step__min_samples_split':[2, 5, 10], 'rf_step__min_samples_leaf':[1, 2, 4],
              'rf_step__max_features':['sqrt', 'log2']}
```

The selected parameters for the Random Forest model strike a balance between complexity, generalization, and computational efficiency. The range for *n_estimators* ensures sufficient trees for stability without excessive computation. *max_depth* values allow exploration of both shallow and fully-grown trees to capture patterns of varying complexity. *min_samples_split* and *min_samples_leaf* balance fine-grained splits with generalization to prevent overfitting. Lastly, *max_features* promotes diversity among trees, enhancing generalization. These ranges effectively optimize performance while avoiding overfitting or inefficiency.

And now let's look at the impact of each parameter :

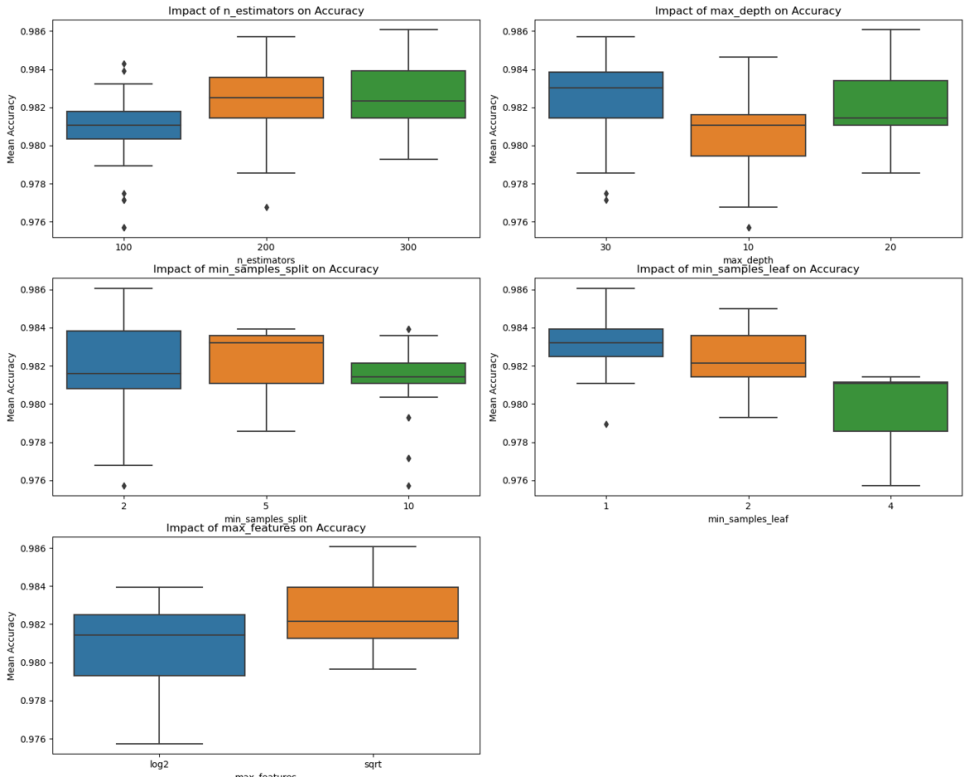


FIGURE 1 – Impact of the hyperparameters on the accuracy

The choice of boxplots was made because they highlight the spread of accuracy scores for each hyperparameter value, giving us a view on both performance consistency and variability.

From the boxplots, we see clear trends in the performance of the hyperparameters. The parameter *n_estimators* shows that accuracy improves as the number of estimators increases, with the highest accuracy and lowest variability achieved at 300 estimators. This makes sense because a higher number of estimators allows the Random Forest model to aggregate more trees, reducing variance and stabilizing performance. For *max_depth*, the results indicate that a depth of 20 provides a good balance between accuracy and stability. While deeper trees, such as those with a depth of 30, still perform well, they exhibit slightly more variability, likely due to overfitting on certain cross-validation folds. This aligns with the best value of *max_depth*=20 found during the search.

The parameter *min_samples_split* performs best with a value of 2, as this allows the trees to grow deeper and capture finer details in the data. However, the increased variance at this setting highlights that the model may occasionally overfit to certain folds. Higher values, such as 5 or 10, reduce this variance but at the cost of lower accuracy. Similarly, *min_samples_leaf*=1 results in the highest accuracy, as smaller leaf sizes allow the model to capture intricate patterns. However, the variance increases, indicating a trade-off between complexity and stability. Larger leaf sizes, such as 4, reduce accuracy and variability but fail to model the data's complexity effectively. Lastly, for *max_features*, the sqrt setting shows slightly better accuracy and lower variance compared to log2, which corresponds with its selection as the best value. The sqrt setting provides a balance between computational efficiency and the diversity of features considered at each split, which helps improve generalization.

Let's also look at the feature importance of our Random Forest :

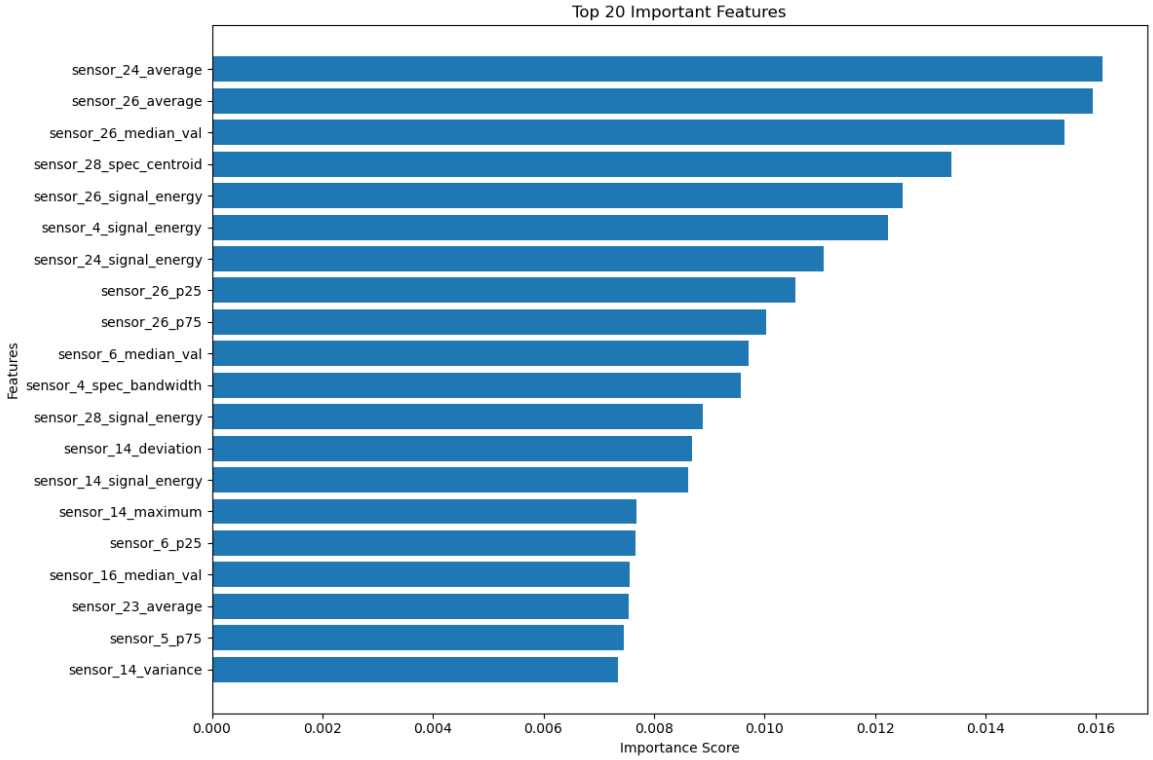


FIGURE 2 – Feature importance : Random Forest

The bar plots further provide insights into feature importance. The complete histogram (14) shows that the majority of features have very low importance, while only a small subset significantly impacts the model's predictions. This corresponds to the behavior of Random Forest, which often relies on a few key features to make accurate predictions. The top 20 features bar plot reveals that features such as *sensor_24_average*, *sensor_26_average*

and *sensor_26_median_val* are the most critical for the model. These features likely capture key patterns or information in the dataset that drive the model's predictive performance. Features lower in importance, such as *sensor_14_variance* and*sensor_5_p75*, contribute less and could potentially be removed to simplify the model without substantial loss of accuracy.

After training and fitting, here are the best values for each hyperparameter :

n_estimators	*min_samples_split*	*min_samples_leaf*	*max_features*	*max_depth*
300	2	1	sqrt	20

With these parameters, we obtain 89,4857% accuracy on the private test.

The results correspond well with the best hyperparameter values found during the `RandSearch` process. The settings align with the trends observed in the boxplots. They give a balance between capturing the complexity of the data and maintaining stability across different cross-validation folds.

To further investigated the precision of the model, a confusion matrix can be plotted in order to display the precision on each class :

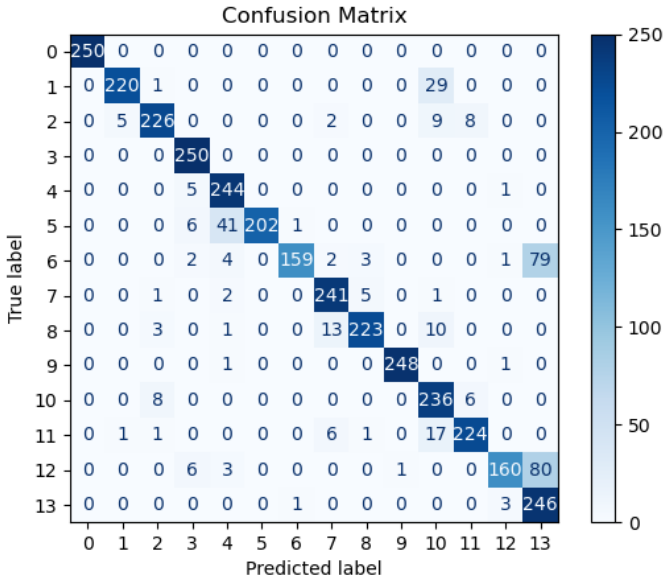


FIGURE 3 – Confusion matrix of the Random Forest

The confusion matrix for the Random Forest model shows that it performs well overall, particularly for classes such as 0, 3, and 9, which are predicted mostly perfectly. Class 0, for example, has all 250 samples classified correctly, and class 9 achieves 248 correct predictions out of 250. However, there are notable issues with specific classes, such as 1, 5, 6, and 12. Class 1 is misclassified as 10 in several instances, and class 6 is frequently confused with 13, 5 with 4 and 12 with 13, suggesting some overlap in features between these classes.

This results suggest that Random Forest is very accurate for most of the classes (and even perfectly accurate sometimes) but has some issues with some classes such as 6 and 12. This tells us that maybe another model should be used if the focus is made on those classes during a classification but from a general point of view, the model is very accurate.

In conclusion, the analysis of hyperparameter impact, feature importance and class prediction confirms that the Random Forest model is well-tuned. The selected hyperparameters provide high accuracy while maintaining stability.

4.2 Perceptron

A Perceptron is a basic linear classifier in supervised learning that models the relationship between inputs and outputs by finding a hyperplane that separates the data into distinct classes. It predicts a class by computing a weighted sum of the inputs and passing it through a activation function to produce an output. The Perceptron learns by iteratively adjusting weights and bias using a simple update rule based on the prediction error.

First we define the domain from which our cross-validation will try the hyperparameters :

```
parameters = {'perceptron_step__penalty': ['l2', 'elasticnet', None],
              'perceptron_step__alpha': loguniform(1e-5, 1e-2),
              'perceptron_step__max_iter': randint(500, 5001),
              'perceptron_step__tol': loguniform(1e-5, 1e-2),
              'perceptron_step__fit_intercept': [True, False]}
```

The selected parameters for the Perceptron model are chosen in a way to cover a large panel of values. The *penalty* options allow exploration of ridge ('l2'), combined L1/L2 ('elasticnet') regularization, and no regularization, providing a balance between sparsity and stability. The *alpha* range ensures meaningful regularization strength, while *max_ iter* and *tol* balance optimization precision and runtime. The inclusion of *fit_ intercept* options allows the model to adapt to datasets with or without centering. Together, these parameters explore a large part of the perceptron's hyperparameter space.

After a search for the best hyperparameters, the following values were obtained :

alpha	*fit_ intercept*	*max_ iter*	*penalty*	*tol*
$1.7019223026554016 \times 10^{-5}$	True	3814	elasticnet	0.004379288100052079

and an accuracy of 67.9714%. The accuracy being so low (compared to Random Forest), the method is considered not suited for our dataset.

To give some credit to the model, the confusion matrix gives some interesting information :

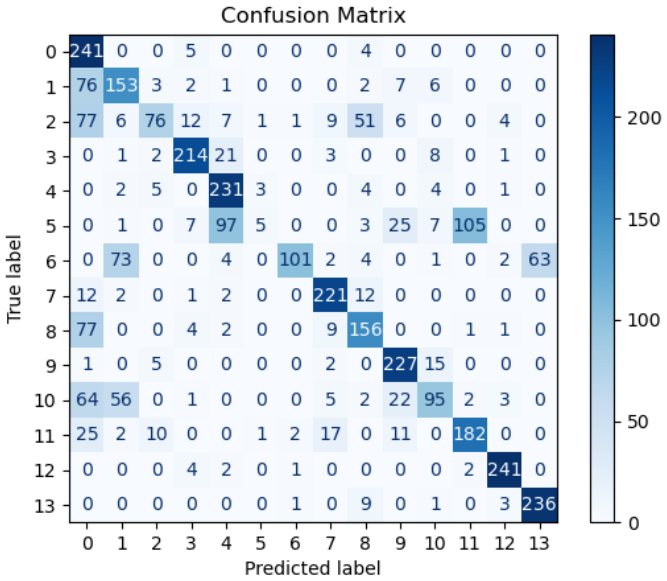


FIGURE 4 – Confusion matrix of the Perceptron

From a general point of view, the model is not very accurate but it presents some good accuracy locally. Classes 0, 4, 12 and 13 present a high precision when predicted. This model could, in theory, be used for predictions if we focus on this classes but, some other models could be as or more precise on these classes while being more accurate in general (like Random Forest).

4.3 Recurrent Neural Networks (RNNs)

Why We Tried RNNs
When it comes to Human Activity Recognition (HAR), time matters. The sequential nature of the data makes Recurrent Neural Networks (RNNs)—and specifically Long Short-Term Memory (LSTM) networks—a natural fit. These models are great at spotting patterns over time, which is key when you're analyzing sensor data. That's why we decided to give LSTMs a shot, hoping they'd help us capture the temporal dependencies in the dataset better than traditional methods.

How It Went
We built and trained a custom LSTM model, but the process wasn't exactly smooth sailing. First, there was a lot of prep work : handling missing values, normalizing the data, and reshaping it so the model could process time steps. While the model initially showed some promise, it ran into trouble as training progressed. Here's how it performed over 10 epochs :

Table 1 – LSTM Training and Validation Performance

Epoch	Train Loss	Validation Loss	Validation Accuracy
1	2.0235	1.3097	54.57%
2	1.0214	1.0259	62.86%
3	0.8316	0.7506	73.57%
4	0.6242	0.7884	76.29%
5	0.8616	0.6715	81.57%
6	0.5356	0.5145	83.43%
7	0.4148	0.3671	89.43%
8	0.4004	0.3872	85.86%
9	0.3506	0.3746	88.71%
10	0.3460	0.7540	74.86%

What Went Wrong
While the LSTM model hit a peak validation accuracy of 89.43% around epoch 7, things went downhill from there. By epoch 10, validation accuracy had dropped to 74.86%, even though the training loss kept decreasing. This was a textbook case of overfitting : the model memorized the training data but couldn't handle new, unseen data.

Here's why this likely happened :

— **Model Complexity :** LSTMs are sophisticated and require fine-tuning to work well. Without precise adjustments, their complexity makes them prone to overfitting.
— **Learning Rate Issues :** We might have used a learning rate that was too high or too low, which could have caused instability during training.
— **Limited Regularization :** We didn't implement techniques like dropout or weight decay, which are key to preventing overfitting in deep learning models.
— **Small Dataset :** HAR datasets aren't huge, so it's hard for complex models like LSTMs to generalize without overfitting.

Confusion Matrix Analysis
The confusion matrix in Figure 5 provides a more better view of how well the model distinguishes between the different activity classes. Each cell in the matrix represents the number of instances from a given true class (rows) that were predicted to belong to another class (columns). A perfectly accurate model would have all values concentrated along the main diagonal, indicating that every activity was correctly classified.

In our case, while certain classes were recognized more accurately (as indicated by relatively high counts along the diagonal), others were frequently misclassified. This discrepancy suggests that the model learned strong temporal patterns for some activities, but struggled with classes that have either very subtle differences or are underrepresented in the training set.

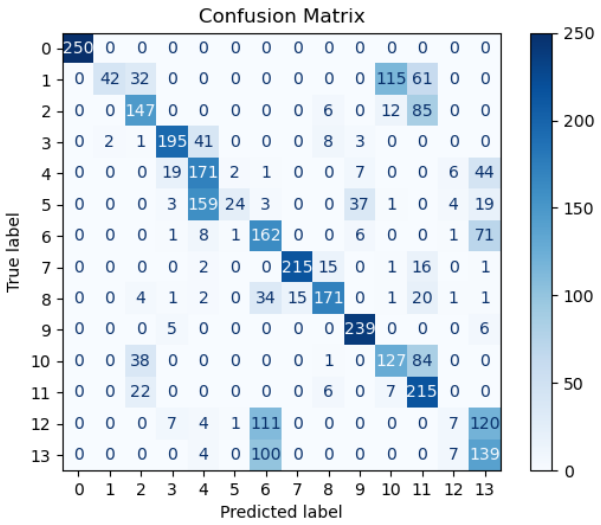


FIGURE 5 – Confusion matrix of the LSTM-based RNN model's predictions.

The Takeaway

In the end, while the LSTM approach was an interesting experiment, it wasn't the best fit for this project. We pivoted back to algorithms we were more familiar with, like Random Forest, which gave us a reliable accuracy of 89.48%. LSTMs have a lot of potential, but to use them effectively, you need deeper expertise in neural networks and more resources. Revisiting them in the future, with more experience under our belt, might be worth it. For now, sticking with interpretable and reliable models was the better call.

4.4 Multi-Layer Perceptron

A Multi-Layer Perceptron is a type of neural network composed of multiple layers of interconnected nodes, where each node, or neuron, performs a weighted sum of its inputs followed by an activation function. It consists of an input layer, one or more hidden layers, and an output layer. The MLP learns by iteratively adjusting its weights and biases using backpropagation, a process that minimizes the error between the predicted and actual outputs.

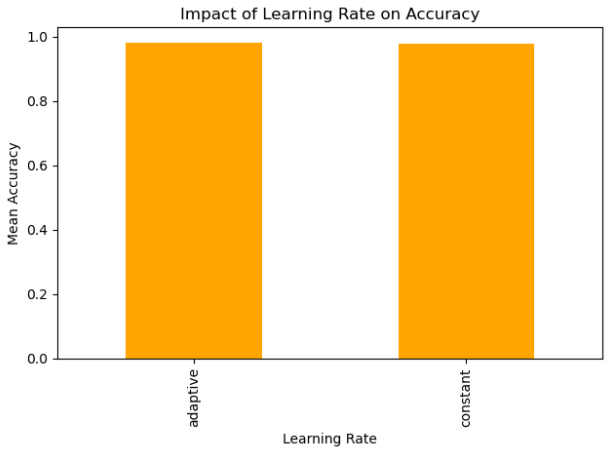
We can define the domain of the hyperparameters used in a MLP as follows :

```
parameters = {'mlp_step__hidden_layer_sizes': [(64,), (128,), (64, 32), (128, 64)],
              'mlp_step__activation': ['relu', 'tanh'],
              'mlp_step__solver': ['adam', 'sgd'],
              'mlp_step__alpha': [0.0001, 0.001, 0.01],
              'mlp_step__learning_rate': ['constant', 'adaptive'],
              'mlp_step__max_iter': [500, 1000, 2000]}
```

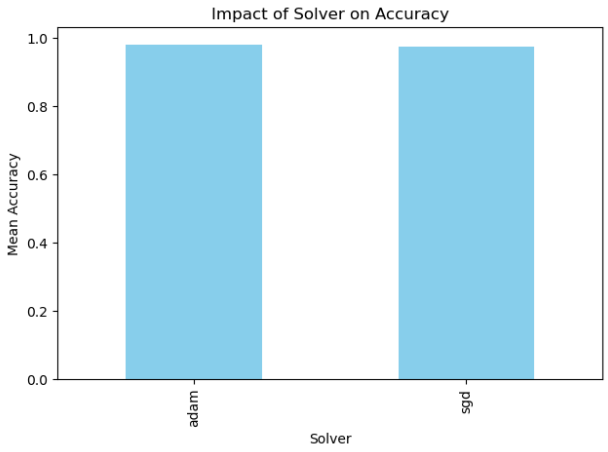
The hidden layer sizes include simple architectures ((64,) and (128,), one layer) and more complex ones ((64, 32) and (128, 64), two layers), covering diverse model capacities. Activation functions balance efficiency (ReLU) with smoothness (tanh). The solvers, Adam and SGD, offer a trade-off between adaptability and simplicity (Adam adapts the learning rate whereas SGD has a fixed one). Regularization strengths (alpha) range from light (0.0001) to moderate (0.01), addressing overfitting risks. Learning rates (constant and adaptive) enable SGD to have some variability in its learning rate, while maximum iterations (500, 1000, 2000) ensure adequate convergence opportunities. These values are computationally practical and provide sufficient coverage of the domain to identify a suitable model for the dataset.

After running the protocol on this model, we obtain an accuracy of 98.86% on the test set. The impact of the different variables can be studied with the following graphs.

Number of iterations and learning rate
The impact of these parameters on the mean cross-validation accuracy is displayed below :



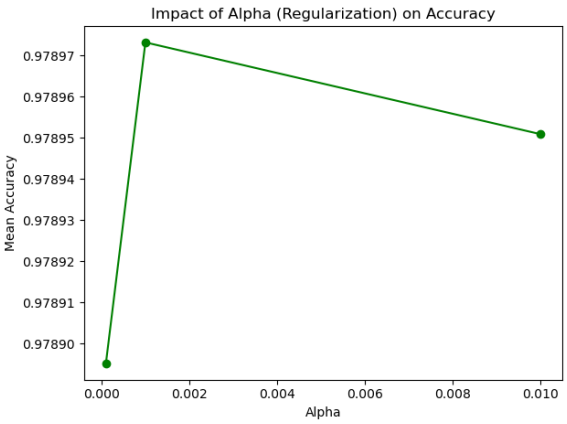
Impact of the type of learning rate



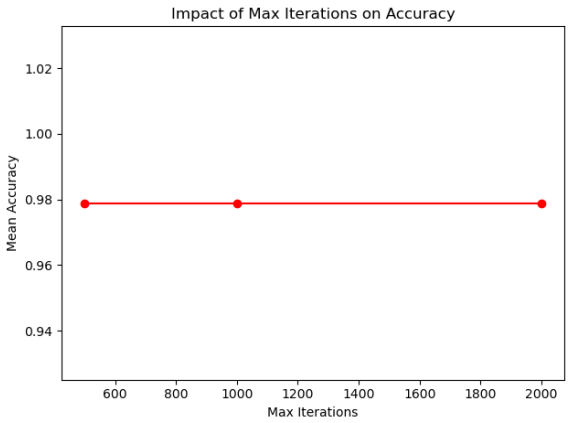
Solver's impact

It can easily be seen that the accuracy doesn't change the accuracy, thus, these two parameters don't have an impact on the accuracy of the model.

Alpha and number of iterations
The impact of these parameters on the mean cross-validation accuracy is displayed below :



Alpha's impact



Impact of the number of iterations

These parameters show distinct patterns : while alpha initially increases accuracy before eventually decreasing it, *max_iter* maintains a constant accuracy across its entire range. A moderate alpha yields the best precision, but the number of iterations does not appear to influence performance. Therefore, the lowest *max_iter* value is preferable to minimize computation time. However, it is worth noting that the variations in accuracy across different alpha values are minimal. As a result, even if one alpha appears slightly better, selecting a different value is unlikely to significantly affect the model's performance.

Hidden layer sizes and activation functions
The impact of these parameters on the mean cross-validation accuracy is displayed below :

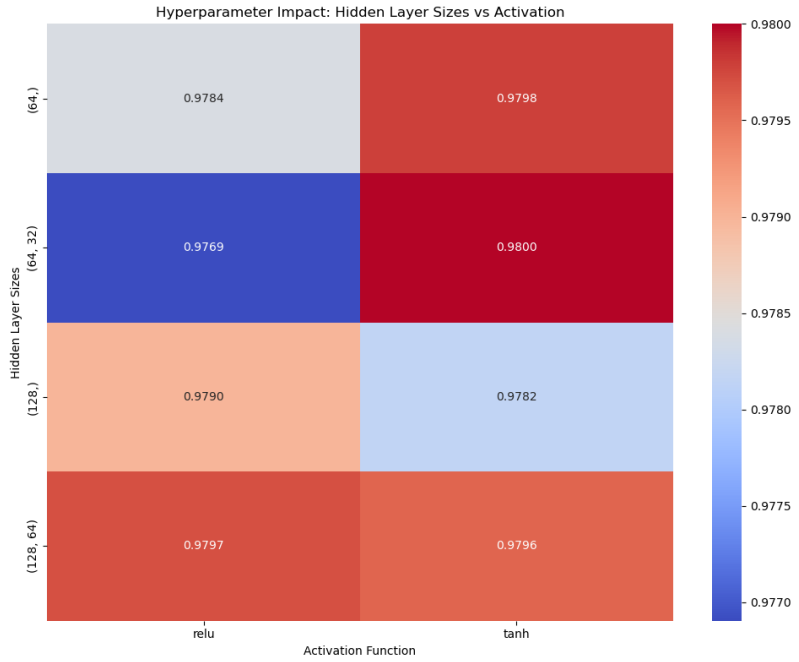


FIGURE 8 – Impact of the hidden layer sizes and the activation functions

The combination (64, 32) with the activation function tanh achieves the highest mean cross-validation accuracy based on the heatmap. However, since this is an average across all cross-validation folds, it does not guarantee that this configuration will perform best on unseen data. Another combination may have slightly lower mean accuracy but exhibit more consistent performance or better results on the final validation set.

It is also important to notice that the difference of precision when selecting different parameters is not extreme (when there is one), meaning that the model's real performance will be seen when tested on the private test.

When tested on the private test, an accuracy of 75.6% was obtained. For more insight of this precision we can analyse the confusion matrix :

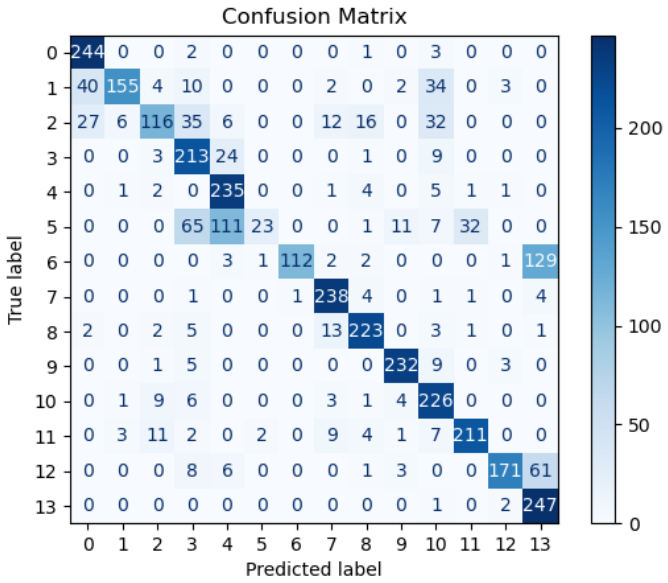


FIGURE 9 – Confusion matrix of the Multi-Layer Perceptron

The confusion matrix shows that the model performs well for most classes, particularly 0, 4, 7, 9 and 13, which are predicted with high accuracy. However, it struggles with classes like 1, 2, 5, 6 and 12. For instance, class 1 is often misclassified as 0 or as class 10, and class 5 is frequently confused with 3 and 4. These errors suggest overlapping features between those wrongly classified classes.

This result was obtained with the following parameters :

solver	*max_iter*	*learning_rate*	*hidden_layer_sizes*	*alpha*	*activation*
adam	500	constant	(128, 64)	0.001	tanh

The protocol appears to have selected hyperparameter values in alignment with the analysis discussed earlier, with the exception of *hidden_layer_sizes*. While (64, 32) achieved slightly higher mean cross-validation accuracy, the protocol likely determined that (128, 64) has the potential for greater precision, possibly due to better generalization on the validation set. Although a larger network like (128, 64) might increase the risk of overfitting, the small difference in mean cross-validation accuracies suggests that this is unlikely to be a significant issue. Furthermore, the choice reflects a balance between capturing complexity in the data and maintaining robustness, as the protocol likely considered performance consistency across folds in addition to mean accuracy.

4.5 Logistic Regression

Multinomial logistic regression, is an extension of logistic regression designed to handle more than 2 outcomes. It models the relationship between a set of explanatory variables and a dependent variable. It predicts the class by computing probabilities for each category and assigning the class with the highest probability.

To analyze the impact of hyperparameters on the model's accuracy, we first define the range of values from which cross-validation will sample during the search process :

```
parameters = {
     'log_reg_step__C': [0.01, 0.1, 1, 10],
     'log_reg_step__solver': ['lbfgs', 'saga', 'sag', 'newton-cg'],
     'log_reg_step__max_iter': [100, 200, 300, 400, 500]
 }
```

The selected parameters for the Logistic Regression model balance the model's performance on the training data, optimization efficiency, and convergence, ensuring a trade-off between simplicity, accuracy, and computational cost.

Now, let's analyze the impact of each parameter :

Regularization strength

As C increases, the mean accuracy improves as well. This indicates that the more the model is allowed to fit the data, the bigger will be the accuracy. Nonetheless, beyond C=1, the improvement in accuracy becomes negligible, and, more outliers appears, suggesting that the model is less stable.

Thus, the best performance is observed at C=1, balancing accuracy and stability.

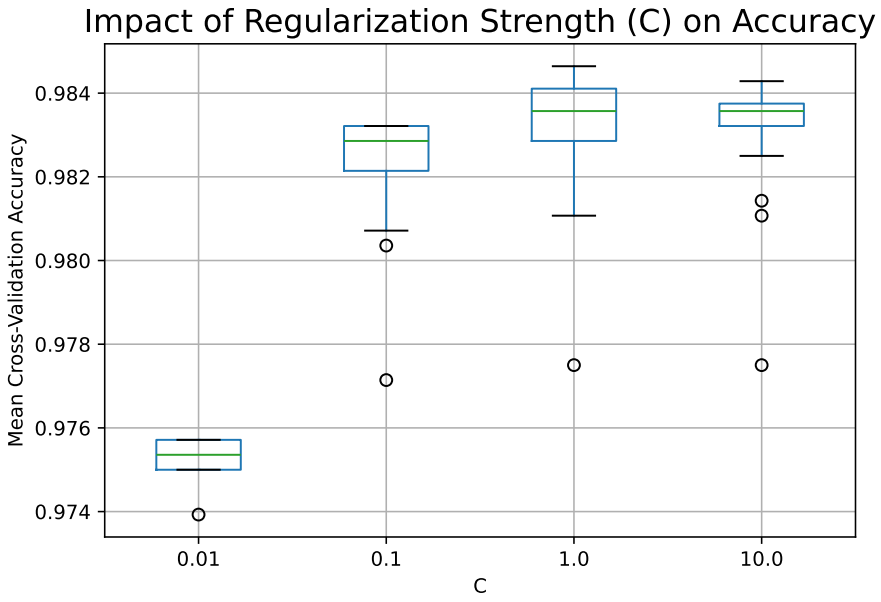


FIGURE 10 – Boxplots of the mean accuracy according to the regularization strength

Solver choice

First of all, to solve this identification problem we use the function `LogisticRegression` from `sklearn.linear_model`, which prvides 6 solvers : lbfgs', 'liblinear', 'newton-cg', 'newton-cholesky', 'sag' and 'saga'. As our problem is multinomial, liblinear is not applicable. Furthermore, during our tests, the newton-cholesky solver did not seem to work as well.

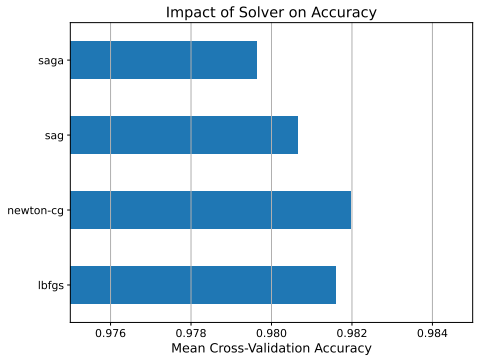


FIGURE 11 – Mean accuracy according to the solver

The bar plot demonstrates that among the tested solvers, the newton conjugate gradient, *newton-cg*, achieves the highest mean accuracy, making it the most suitable solver for our problem. While *lbfgs* also performs well, the other are clearly behind, highlighting the importance of selecting the appropriate solver to maximize model performance.

Maximum number of iterations

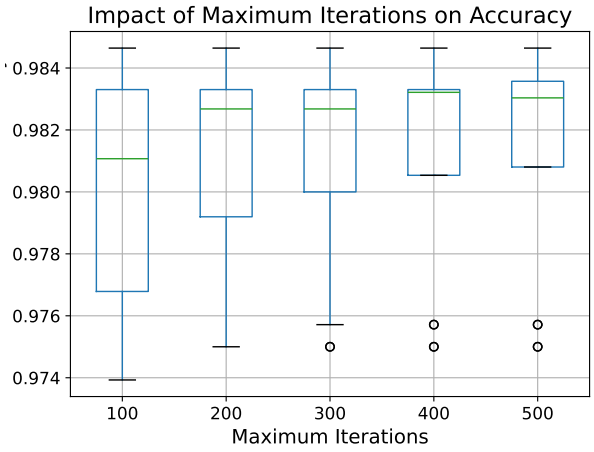


FIGURE 12 – Mean accuracy according to the solver

Beyond 200 iterations, the accuracy shows little to no improvement, suggesting that additional iterations are not necessary for the solver to converge to optimal values.

As the number of iterations increases, the variance decreases ; however, outliers become apparent beyond 200 iterations.

While 200 iterations might seem like a reasonable choice, the maximum number of iterations will be set to 100 for computational efficiency, as there is no significant difference in accuracy.

After training and fitting, here are the best values for each hyperparameter :

Regularization strength (C)	Solver	Maximum number of iterations
1	'newton-cg'	100

With these parameters, we obtain 74.4857% accuracy on the private test.

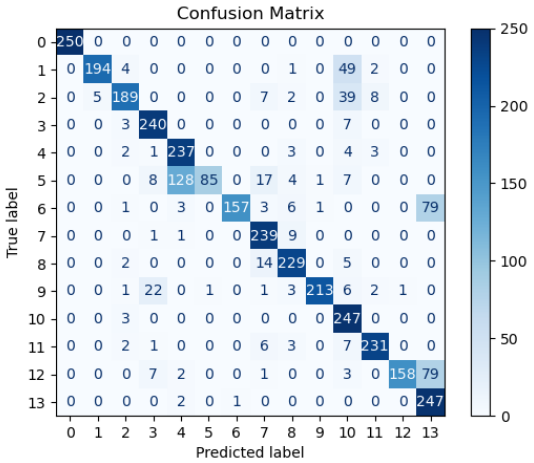


FIGURE 13 – Confusion Matrix of the logistic regression model

The model appears to struggle with distinguishing rope jumping and running from playing soccer, but only in one direction, as it rarely confuses soccer with these activities. Similarly, the model has difficulty differentiating sitting and standing from ironing. Another insight is that the model frequently predicts Nordic walking as normal walking rather than recognizing it correctly. Nonetheless, the majority of the predictions are concentrated along the diagonal, indicating that the model correctly classified a large portion of the samples. This demonstrates that the model performs well overall.

In conclusion, logistic regression performs adequately, but other methods yield better results. Since logistic regression relies on the logit function, the model is forced to make a definitive choice. However, this can be challenging when the differences between certain activities are minimal.

4.6 Gradient boosting

Gradient Boosting is an ensemble learning technique that builds a strong predictive model by sequentially combining weaker models to minimize the residual errors. It optimizes model predictions through the gradient of the loss function, improving performance iteratively. This method is highly effective at capturing complex patterns in data and enhancing accuracy, while its iterative nature ensures robustness and adaptability.

In this case, the Gradient Boosting Classifier method from `sklearn.ensemble` was used, combining regression trees as the base weak learners.

The search for optimal parameters was challenging because we used the exact method, which is not well-suited for large datasets. As a result, we decided not to pursue further investigations into this model. Additionally, its performance on the test set did not seem to surpass the regression trees model, and the model's computational time was prohibitively high.

Nonetheless, the table below presents the optimal parameters that achieved an accuracy of 83.4571% on the test set :

n_estimators	subsample	max_depth	min_samples_leaf	Max_features
400	0.6	40	191	'sqrt'

5 Selecting the best model

In order to select the best model, let's first summarize the computation time and accuracies on the private test :

	Random Forest	**Perceptron**	**Multi-Layer Perceptron**	**Logistic Regression**	**XGBoost**	**RNN**
Computation time (sec)	0.07	0.01	0.02	0.01	0.40	2.17
Accuracy (%)	89.4	67.9714	75.6	73.1714	83.4571	60.1143

Based on these results, the RNN and XGBoost models are the most computationally expensive, with RNN requiring significantly more time than the others. However, XGBoost achieves a good accuracy of 83.46%, making it a strong candidate for tasks where accuracy outweighs computation time. On the other hand, models like Perceptron, Logistic Regression, and Multi-Layer Perceptron (MLP) are very efficient in terms of computation time but do not achieve high accuracy, with values of 67.97%, 73.17%, and 75.6%, respectively. Among all models, Random Forest strikes the best balance, achieving the highest accuracy (89.4%) while maintaining a low computation time of 0.07 seconds. Considering both efficiency and performance, Random Forest emerges as the best choice for this dataset.

6 Summary

This project focused on Human Activity Recognition using time-series data collected from 31 sensors. The goal was to preprocess the data, establish a robust experimental protocol, and evaluate various machine learning models to identify the most effective and efficient solution for the task.

Data preprocessing involved handling missing values using conditional and unconditional mean imputation. Features such as mean, variance, skewness, and spectral properties were extracted from the time-series data to simplify the raw information and enhance model performance. The processed data was divided into training, validation, and test sets, with a consistent protocol applied across all models to ensure fair evaluation.

A range of machine learning models, including Random Forest, Perceptron, Multi-Layer Perceptron (MLP), Recurrent Neural Networks (RNNs), Logistic Regression, and Gradient Boosting (XGBoost), were tested. Each model underwent hyperparameter optimization using `RandomizedSearchCV`, and their performance was assessed based on accuracy and computation time. Random Forest emerged as the best-performing model, achieving 89.4% accuracy with a computation time of just 0.07 seconds.

The analysis of confusion matrices provided insights into each model's performance. Random Forest excelled in accurately classifying activities like "Lying," "Standing," and "Walking very slow," with minimal misclassification. However, it showed struggles with overlapping activities, such as "Nordic walking" vs. "Normal walking", "Running" vs. "Playing soccer" and "Rope jumping" vs. "Playing soccer" likely due to sensor similarities between these activities.

Other models exhibited varying degrees of success. RNNs, despite being suited for time-series data, struggled with overfitting and failed to distinguish between most of the activities, like "Running" and "Playing soccer". MLP and Logistic Regression, which have comparable accuracies, exhibit similar confusion matrices. While some activities are classified accurately, many are only moderately or poorly classified. For instance, "Rope jumping" is frequently mistaken for "Playing soccer", a recurring issue across multiple models, alongside the common confusion between "Normal walking" and "Nordic walking". The Perceptron falls between RNN and these two models in terms of performance, with a few activities being well classified but the majority showing significant misclassification. Finally, Gradient Boosting (15) demonstrates performance close to Random Forest but with more inaccuracies in certain classes.

In conclusion, Random Forest was selected as the best model for this task due to its strong balance of accuracy and computational efficiency. While other models either lacked accuracy or were computationally intensive, Random Forest proved to be a reliable choice for this complex time-series classification problem. This project underscores the importance of evaluating both performance and efficiency when selecting machine learning models for real-world applications.

7 Appendix

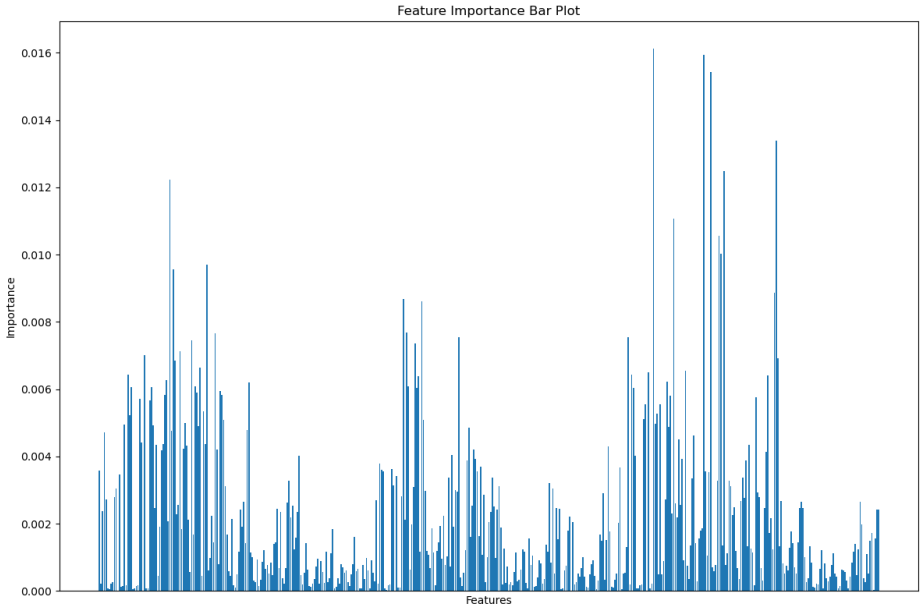


FIGURE 14 – Random Forest : Importance of each feature

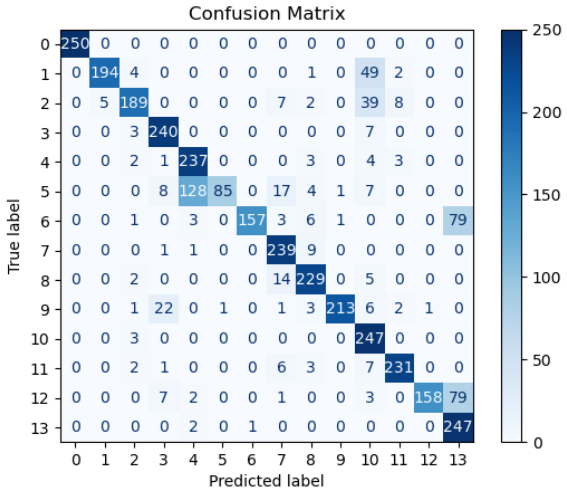


FIGURE 15 – Gradient Boosting confusion matrix