Smart Assistant Interaction through Visual Gesture Recognition using a Kinect Sensor

Enrico Buratto
Department of Mathematics
University of Padua
Padova, Italy
enrico.buratto.1@studenti.unipd.it

Mariano Sciacco
Department of Mathematics
University of Padua
Padova, Italy
mariano.sciacco@studenti.unipd.it

***Abstract*—In the last decade, the raise of smart assistant devices played a key role for the everyday life as they help to retrieve information from the web, manage other home devices and execute routines independently. Even though the vocal interaction is getting more and more efficient, not always the commands are interpreted correctly. Moreover, looking at deaf people and/or people with speech disabilities, the effective interaction is drastically reduced, thus limiting the use of this kind of devices. The main goal of this project is to use the visible body parts of a person to interact with smart devices like Google Home through human pose estimation and gesture recognition. During our work we compared two different solutions – Google MediaPipe and NiTE2 middleware for Kinect –, and for both of these use-cases we tried different classification algorithms: *multinomial logistic regression*, *ridge regression*, *random forest*, *support vector machines* and *multilayer perceptron*. Through intensive trial and error experiments we were able to get F1-scores in a range between 0.71 and 0.97 for the different use-cases and classification algorithms, with an average accuracy never below 0.70.**

***Index Terms*—human pose estimation, gesture recognition, Kinect, Google smart assistant, MediaPipe, NiTe2, OpenCV**

I. Introduction

In this period of time, smart assistants are getting more and more in use especially to manage home devices such as lights, smart plugs, thermostats and even smart ovens or coffee makers. These kind of technologies aim at simplifying the everyday life so that the vocal interaction to execute commands does not need the use of a smartphone or a remote to manually interact every time. Obviously this may not be achieved for deaf people and/or people with speech disabilities, as the effective interaction is limited, thus having to use touchscreens or remotes instead. On the other hand, we thought that it would be interesting to maintain a device-less interaction by introducing visual gesture recognition based on trained models of gestures that the user can replicate in order to interact with smart devices and assistants.
Human pose and gesture recognition are well-known problems in the Machine Learning field: a lot of studies have been done and the topic is still under heavy development. Our work was to address this problem using and comparing different libraries (*i.e. MediaPipe, OpenNI/NiTE2*), devices (*i.e. simple webcam, Microsoft Kinect*™ [1], [2]), approaches (*e.g. existing datasets* or *personally-built datasets*) and classification algorithms (*Scikit-learn's* [3] *logistic regression, ridge regression, random forest classifier, support vector machines and multi-layer perceptron*) in order to apply them in a real-life scenario. At the end of the experiments, we finally achieve overall good F1-scores and good accuracy and recall measurements. Despite this, our predictions are particularly subject to overfitting, especially using the most sophisticated classifiers, due to the natural difficulty of getting a coherent and well formed dataset; anyhow, this will be analyzed more in deep in the next sections.

II. Related Work

As previously said, human pose, gesture and action recognition problems are acclaimed in the computer vision field: many researches have been done and plenty of literature have been produced. The reader may find useful to analyze some work related to our research such as [4], [5], [6]. For a more detailed description of MediaPipe, the reader may refers to [7]; the MediaPipe libraries we used are only body pose related, therefore reader may refer to BlazePose official paper [8]. Since there's not many literature describing NiTE2 functionalities, reader may find useful to compare our approach to these different applications of computer vision involving the Microsoft Kinect™ [9], [10], [11] and visit the deprecated OpenNI website [12].

On a real app implementation point of view, our approach is similar to the Google project called Soli [13], [14], that tracks gestures such as hands waving in order to execute commands in front of a display. In this case, Google planned on using a specific chip (*Soli radar chip*), which is low powered and based on motion detection, thus allowing it to be always active to track subjects movements. This chip is integrated in smart displays like the Google Nest Hub [15]. Nevertheless, our project differs from the hardware and interaction perspectives. More specifically, rather than interacting with a gesture to perform actions related to what is on the screen, we want to assign a single gesture to a specific action without having to use a display. This way, while acquiring a real-time video through the Kinect™, the data is immediately elaborated to invoke the Google Assistant whenever needed.

III. Dataset

The data used for this project is mainly based on two datasets: a custom made dataset of poses and an external dataset called APE [16]. Each dataset has been used separately depending on the library in use, so that we could compare results based on the approach adopted. We also had to apply preprocessing and data augmentation in order to manipulate each image before being classified.

A. APE & Custom-made Dataset

The APE dataset contains several human poses for 3D pose estimation with RGB images, as well as depth maps. Inside there are also many different types of actions performed in a unconstrained environment and with different camera perspectives that are enough challenging in order to obtain a good pose estimation in distinctive conditions. The images in this dataset are around 24,000 with 245 sequences made from 7 different subjects performing 7 different categories of actions (*e.g.* balance, wave and clap).
As well as having used a pre-existing dataset we also tried to build a custom dataset. The data have been acquired via a Microsoft Kinect™ following a procedure defined by us: for every gesture to acquire, we save the body keypoints coordinates of ten seconds of real-time video into two different *csv* files, one for the MediaPipe pipeline and one for the NiTE2's one. We were able to collect 6 different subjects performing 7 poses in the same situations. Each subject had different body shapes and characteristics, and thanks to the data augmentation phase we were able to make the dataset varied.

B. Image Filtering and Preprocessing

Regarding the images and the real-time videos, for each frame we had to execute preprocessing in order to convert it from BGR to RGB. In fact, we apply this kind of preprocessing in different phases, where we had to acquire the Kinect™ video flow, but also before processing the images from the APE dataset. Besides this conversion, for the Kinect™ we also needed to resize each frame acquired in order to have an exact height and width to work with; the affine transformation we needed was the following:

$$\begin{bmatrix} x'_i \\ y'_i \\ z'_i \end{bmatrix} = 1.25 \cdot \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} \tag{1}$$

Moreover, we also had to reorder into a `uint16` array the output obtained in order to match the depth data retrieved by the Kinect with OpenCV [17], that is the library we used for video and image manipulation.

C. Data Augmentation

Since making a custom dataset can lead to very inaccurate results, due to the scarcity of valid frames, we decided to augment the data acquired by us by introducing horizontal flipping and small random resizing as referred in [18] and shown respectively by these two affine transformations:

$$\begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{2}$$

$$\begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \begin{bmatrix} rand(0.9, 1) & 0 \\ 0 & rand(0.9, 1) \end{bmatrix} \times \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{3}$$

In addition to this, we also introduced image darkening and brightening, in order to simulate different environment lights conditions. These techniques can help to improve the model and reduce overfitting, and in fact led us to more precise estimations.

IV. Method

In order to achieve the initial goal, we developed a Python application made of different components: the Kinect™ camera (hardware component), the client (with all the business-logic for gesture recognition) and the server (used to interact with external APIs). In details, we developed the gesture recognition client that uses the Kinect™ in order to capture real-time videos using the depth camera. The client analyzes each frame and, once a gesture is recognized and confirmed, a message is sent to the server to act as a gesture translator, thanks to which it is possible to invoke the Google Assistant API, so the action is executed (*e.g.* turning on lights). As you may see from figure 1, the workflow between the components is overall straight-forward with the exception of the gesture recognition part.

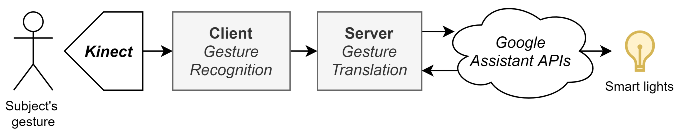

Fig. 1. Architecture workflow of the application.

Regarding the client part, we started our development with a simple 720p@20fps camera and trying to get all the keypoints that *MediaPipe Holistic* is able to recognize. This library can identify up to 33 body landmarks, 21 hand landmarks each and 468 face landmarks. In order to do that it uses different techniques *under the hood*; this goes beyond our discussion, but reader may find useful to investigate the matter further from these papers [19], [20].
At first, we attempted to train a model with the random forest classifier supplied by Scikit Learn Python library. However, the model was quite inaccurate since MediaPipe uses several landmarks for the face, thus these obviously weights more in the model rather than the other keypoints for the body part. Due to this, we were unable to obtain a correct gesture classification with multiple classes and with different subjects. Therefore we decided to not include the face keypoints, as well as the hands keypoints, so that we were able to use only the remaining pose keypoints. Fixed this, we moved on to define the overall architecture of our method (figure 2).

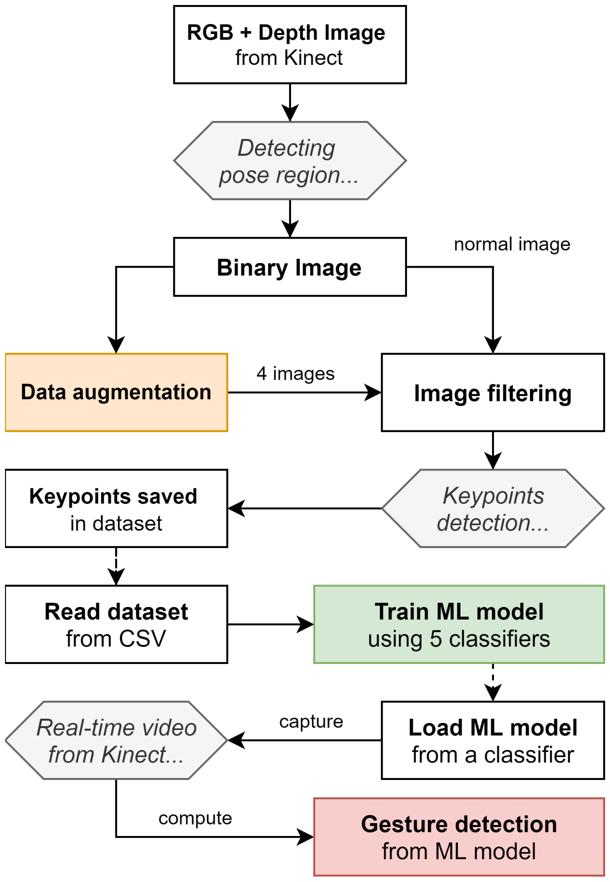

Fig. 2. Gesture recognition flow of the application.

A. Training and Gesture Recognition

Regarding the training part, once the model was corrected we used five different classifiers natively offered by Sklearn:

- *Logistic regression*: this is one of the simplest classification algorithms. It is typically used in binary classifications (*i.e.* ”yes/no” decision), but it can be expanded to work with multiple nominal categories; in this case, it’s called multinomial logistic regression, and since we have 7 different classes we obviously use that;
- *Ridge regression*: this is a linear model, since it’s an extension of linear regression that adds a L2 regularization to the loss function; this means that a term is added to the loss function in order to avoid overfitting. Being this a regression model it should not work for our use-case, but Scikit learn’s implementation offers a multiple class classification exactly like in the logistic regression’s case;
- *Random forest*: this is a decision tree algorithm that build a large number of decision trees from bootstrap samples from the dataset; at each split point, it also selects a subset of features. The latter is particularly important in our goal to reduce overfitting, since a higher number of features assigned to each tree is directly related with the grow of the overfitting phenomena;
- *SVM*: support vector machines is a typical classification algorithm based on hyperplane construction. The algorithm puts every data point in a hyper-dimensional space and then tries to find the best division of it in an iterative manner; the division that minimize the support vectors sum, *i.e.* the datapoints closer to the hyperplane, is the classificator returned by the algorithm;
- *MLP*: multilayer perceptron is one of the simplest deep neural networks. In Scikit’s implementation, it consists in three layers of neural nodes: an input layer, a hidden layer and an output layer. The algorithm trains iteratively: in fact, at each iteration it computes the partial derivatives of the loss function with respect to the model parameters; it then update the parameters and repeat.

For each of these algorithms we trained a different model, that is subsequently saved in a binary file for later loading and usage. From now on the user can run the program loading the preferred model. During the running phase, the keypoints are detected in real-time through the desired library (MediaPipe or NiTE2), and the program apply the model to predict the class of gesture performed by the user.

B. Gesture Translation

During the real-time video analysis, once the gesture is detected and the relative precision is calculated, we gather each result from a video frame inside a buffer and then, every time a new result is inserted, we check if the buffer has an adjacent subsequence of a gesture registered that satisfies two thresholds, corresponding to the minimum number of repetitions and the minimum average precision required. This is necessary in order to have at least 1 or more seconds of recognition before identifying, as such, that precise gesture in output. Once the gesture is confirmed, there is a configurable cool-down phase where the following frames do not get saved inside the buffer, for a precise number of seconds. This way, the gesture confirmed is then sent through a socket to a server listening to messages and working independently from the main program. The server translates the gesture recognized into a natural language phrase based on a JSON configuration file. After that, the server invokes the Google Assistant using the phrase translated, and in the end the command is performed.
The buffer control sequence is structured as follows:
Let M be the buffer containing the gesture g (and the corresponding precision p) registered at the i frame of the real-time video

$$M = \{\dots, (g_{i-2}, p_{i-2}), (g_{i-1}, p_{i-1}), (g_i, p_i)\}$$

Let N be a subset of M so that $N \subseteq M$ and let $m = |M|$ and $n = |N|$ with $n \le m$; N is a sub-sequence of M where each gesture is adjacent to the other or is far at most 5 positions in the buffer.
This check is important in order to avoid false positive reporting from the running analysis. In the end N is a subsequence with the most number of adjacent repetitions (or with the best average precision) satisfying the thresholds required.

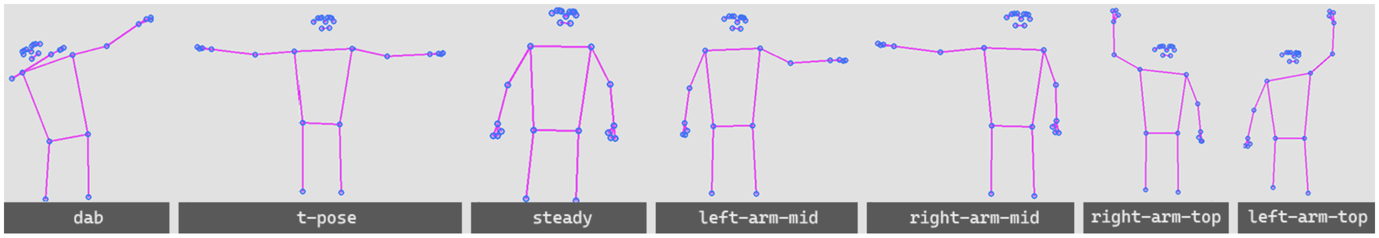

Fig. 3. Gestures used for the custom dataset

V. Experiments

A. Gesture Acquisition

The gesture acquisition process uses the Kinect™ in order to detect all the keypoints of the body, especially the one used for arms and chest. This process allowed us to identify 7 different custom-made gestures that are associated to a particular action (*e.g.* raising up a hand turns on a smart light). These gestures differs from the APE dataset as we wanted to try different poses using MediaPipe and NiTE2. In fact, as the reader may see from figure 3, the gestures are the following: dab (both arms stretched at 45 degrees to the top), t-pose (both arms stretched horizontally), steady (normal pose standing), left-arm-mid (only left arm stretched horizontally in mid air), right-arm-mid (only right arm stretched horizontally in mid air), right-arm-top (only right arm stretched vertically), left-arm-top (only left arm stretched vertically). We chose these gestures since they are classifiable in 2 categories: symmetrical gestures and non-symmetrical gestures. The symmetrical gestures are the steady and the t-pose gesture, while the remaining ones are non-symmetrical. This way we were able to test the accuracy and the other precision measurements for each of them after the data augmentation process. Moreover, regarding the dab gesture, we decided to not differentiate the *left dab* from the *right dab*, thus achieving the same gesture recognition in both directions. As a matter of fact, thanks to the data augmentation process we flip the image horizontally, so that the keypoints of one direction are also saved for the other direction automatically. The flip is only performed for the symmetrical gestures, as well as the dab gesture, otherwise some keypoints may invalidate the non-specular gestures.

B. Training Phase

For the training process we did some trial and error to appropriately configure the models to train. For different models we made use of different parameters; in particular:

- For *Logistic Regression* we modified the inverse of regularization strength (`C`) and the maximum number of iterations (`max_iter`). For different datasets we used the same number of iterations (150) and different `C` values, which are 10^{-5} for our custom dataset with MediaPipe, 10^{-2} with our custom dataset with NiTE2 and 10^{-4} for the APE dataset;
- For *Ridge Regression* we modified the regularization strenght (`alpha`). For our dataset with MediaPipe we used `alpha` = $5 \cdot 10^5$, for NiTE2 `alpha` = 10^2 and for APE dataset `alpha` = $5 \cdot 10^4$
- For *Random Forest* we modified the number of estimators (`n_estimators`), the maximum tree depth (`max_depth`) and the minimum sample leaves (`min_sample_leaf`). For different datasets we used different values, which are:
  - For our custom dataset with MediaPipe: `n_estimators` = 100, `max_depth` = 1 and `min_sample_leaf` = 10;
  - For our custom dataset with NiTE2: `n_estimators` = 30, `max_depth` = 3 and `min_sample_leaf` = 10;
  - For APE dataset: `n_estimators` = 100, `max_depth` = 5 and `min_sample_leaf` = 10;
- For *SVM* we modified the maximum number of iterations (`max_iter`) and the inverse of regularization strength (`C`). For different datasets we used the same number of iterations (200) and different `C` values, which are $5 \cdot 10^{-2}$ for our custom dataset with MediaPipe, 10^{-1} for our dataset with NiTE2 and 2 for APE dataset;
- For *MLP* we modified the regularization strength (`alpha`) and the size of the hidden layers (`hidden_layer_sizes`). For different datasets we used the same hidden layers size (32) and different `alpha`: 10^2 for our custom dataset with MediaPipe, 2 for our dataset with NiTE2 and 4 for APE dataset.

As just stated, we trained our model different times in order to find a good trade-off between the optimal model and a too overfitted model. In order to assess the accuracy of our model we use precision (4), recall (5), F1 score (6) and Sklearn's accuracy (7) [21]. These four metrics are defined as follows:

$$precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \tag{4}$$

$$recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \tag{5}$$

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{6}$$

$$accuracy(y, \hat{y}) = \frac{1}{n_{samples}} \cdot \sum_{i=0}^{n_{samples}-1} \mathbb{1}\left(\hat{y}_i = y_i\right) \tag{7}$$

We applied these measurements on a test split of 30%-70% (30% for the test and 70% for training); to do that we used Scikit utils library, that divides the dataset in a randomized way.
As an additional indicator, we did a qualitative evaluation of the model accuracy using two types of diagrams: confusion matrices, that graphically shows precision and recall measurements, and learning curves, that shows the difference of accuracy of the model for different input sizes.

C. Experimental Results

We ran our experiments on a Ryzen 9 3900X CPU (12 cores/24 threads), 32 GB of DDR4@3200MHz RAM and an AMD Radeon RX570 GPU; with this hardware, we were able to get an acquiring speed of $\sim$15 FPS, about the 75% of the real camera acquiring speed. Our program runs on Python 3.9, with OpenCV 4.5, Scikit-learn 0.24, MediaPipe 0.8.5 and OpenNI 2.2.
After training our models, we were able to recognize the gestures with both MediaPipe and OpenNI/NiTE2 by sending as input the video from the Kinect™. Looking at our experiments the output obtained using MediaPipe seemed overall better than OpenNI/NiTE2, since under the same classification model the calculated probabilities of every gesture were higher most of the times.

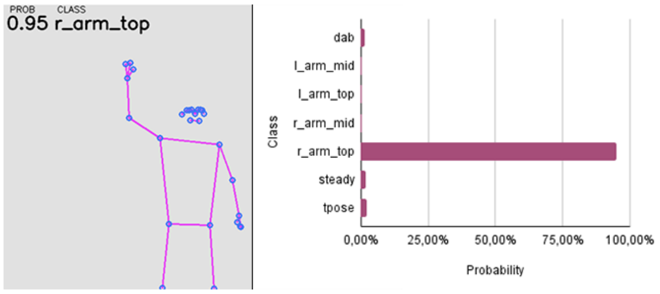

Fig. 4. Gesture recognition using MediaPipe

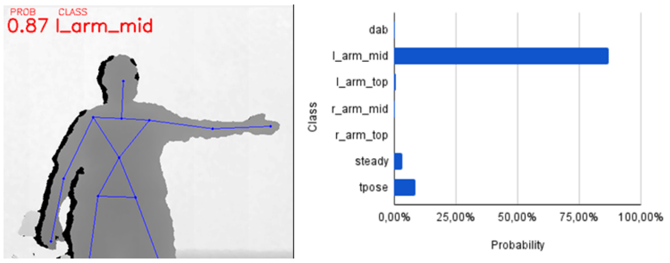

Fig. 5. Gesture recognition using NiTE2

Using our Python application, we could see the video stream in real-time with the keypoints of the subjects recognized by the algorithm chosen. For example, using MediaPipe we decided to track the body keypoints as shown in figure 4; on the other hand, using NiTE2 the procedure was similar, except for the output where we used a depth map (figure 5) so that we could also see the object changing the shades of gray in the background using the IR sensor. On each frame the trained model calculates the percentage of probability for each class of gesture; the graph near each gesture is a corresponding example of that frame analysis. The best fitting gesture we found is the *dab* as it involves several keypoints from the chest and above. Nevertheless, the models had some problems recognizing the difference between the t-pose and the left/right-arm-mid since the keypoints involved are similar, with the exception of the few of the other arm.

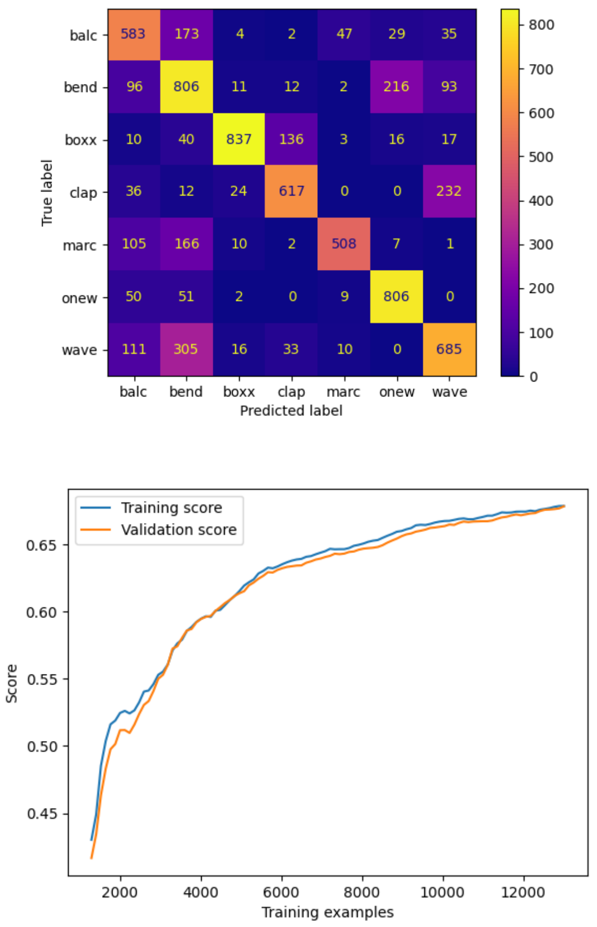

Fig. 6. Confusion matrix and learning curve for Logistic Regression on APE dataset.

Using our custom dataset for model training we got the best results using Logistic and Ridge regression, while on the APE dataset the best outcomes appeared to result using more sophisticated models such as Support Vector Machines and Multilayer perceptron.
The precision, recall, F1-score and accuracy results are shown in table I; these measurements show that certain models (Random forest, SVM and primarily MLP) are subject to overfitting using our custom dataset. This is attributable to the dataset, and

	Custom dataset - MediaPipe				APE dataset - MediaPipe				Custom dataset - NiTE2			
Method	**P**	**R**	**F1**	**A**	**P**	**R**	**F1**	**A**	**P**	**R**	**F1**	**A**
Logistic regression	0.95	0.95	0.95	0.95	0.72	0.70	0.71	0.70	0.91	0.91	0.91	0.91
Ridge classifier	0.94	0.94	0.94	0.94	0.72	0.70	0.70	0.70	0.85	0.82	0.80	0.82
Random forest	0.94	0.93	0.93	0.93	0.84	0.83	0.82	0.83	0.98	0.98	0.98	0.98
SVM	0.85	0.85	0.85	0.85	0.87	0.87	0.87	0.87	0.77	0.76	0.75	0.76
MLP	0.97	0.97	0.97	0.97	0.72	0.71	0.71	0.71	0.78	0.85	0.81	0.85

TABLE I
OVERVIEW OF OUR RESULTS. P: PRECISION, R: RECALL, F1: F1 SCORE, A: ACCURACY

before that to the data acquiring process: even if it has been built to keep data integrity (*i.e.* the collected data for each class has the same size and the number of features must be the same) and to prevent overfitting via data augmentation, we can't say that it contains both enough and accurate data to fully prevent this phenomena. This hypothesis is supported by the results we got using the APE dataset: the accuracy measurements are in fact lower, though the experiments demonstrated that the recognition was more precise in every tested environment.

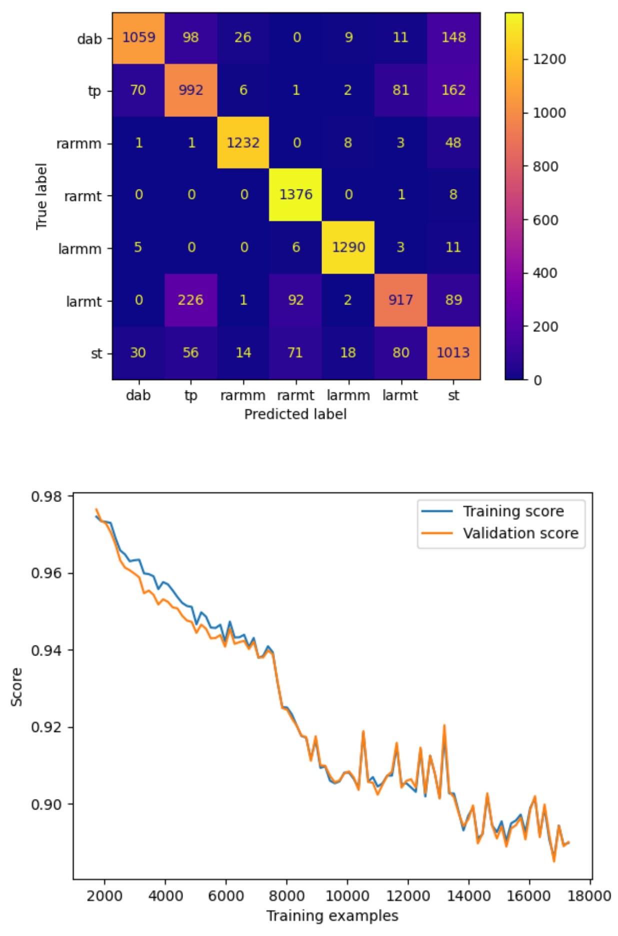

Fig. 7. Confusion matrix and learning curve for SVM on custom dataset.

A last indicator of overfitting are the confusion matrices and, even more, the learning curves. Figure 6 shows a model with none or little overfitting (*i.e.* Logistic Regression on APE with MediaPipe), while figure 7 shows an example of overfitting (*i.e.* SVM on our custom dataset with MediaPipe).

VI. CONCLUSION

The program we developed should be considered more as a *proof-of-concept* than a complete and usable application. The goals of our research was indeed to demonstrate that the technologies we have implemented can be used in such a way, and to compare different machine learning algorithms and approaches in the computer vision field.
With this project, we learned how to develop a program that would apply to a real-life scenario that would include computer vision in order to solve the problem. To achieve this, we had to understand how to use several state-of-the-art libraries such as MediaPipe, OpenCV and Scikit-learn, thus applying machine learning algorithms and the knowledge acquired from the course. Moreover, we also built a skeleton of a possible client-server architecture that can be extended in the future. We can say we are pleased of what we implemented, since we satisfied all the goals we have set ourselves, though more future work can be done.

A. Future Work

Although our program is a fully functional application, it can be extended both for a real world context and as a baseline for different applications. In particular, to make it truly usable we thought that the following features and/or improvements can be added:

- *Time-sensitive gesture recognition*: a big improvement that can be done is to acquire the data and train the model in a time-sensitive manner; this way the program should recognize a complete human action by capturing the movement phase;
- *Multi-phase recognition and GUI*: in order to improve the usability and the learning curve for the user, multiple phases of body pose recognition can be implemented taking advantage other technological resources such as the Google Nest's screen or a Google Chromecast. A good pipeline in this case could be to use different trained models, one for each recognition phase, in order to improve precision and recall.

However, this is not the only applicable solution as this kind of interaction may also be re-used in flights totem or fast-food restaurants where you can use a touchscreen panel to make an order. Especially during a pandemic, where touching objects is not always safe, pose estimation and gesture recognition might be a valuable alternative to classic touchscreens.

References

[1] Z. Zhang, ”Microsoft Kinect Sensor and Its Effect,” in IEEE MultiMedia, vol. 19, no. 2, pp. 4-10, Feb. 2012, doi: 10.1109/MMUL.2012.24

[2] M. Tölgyessy.; M. Dekan; L. Chovanec; P. Hubinský, ”Evaluation of the Azure Kinect and Its Comparison to Kinect V1 and Kinect V2”, 21, 413, 2021, doi: https://doi.org/10.3390/s21020413

[3] G. Varoquaux, L. Buitinck, G. Louppe, O. Grisel, F. Pedregosa, and A. Mueller. ”Scikit-learn: Machine Learning Without Learning the Machinery.”, GetMobile: Mobile Comp. and Comm. 19, 29–33, 1 January 2015, doi: https://doi.org/10.1145/2786984.2786995

[4] J. Shotton et al., ”Real-time human pose recognition in parts from single depth images”, CVPR 2011, 2011, pp. 1297-1304, doi: 10.1109/CVPR.2011.5995316

[5] C. Liu, T. Szirányi ”Real-Time Human Detection and Gesture Recognition for On-Board UAV Rescue.”, 21(6):2180, 2021, doi: https://doi.org/10.3390/s21062180

[6] G. Paraskevopoulos, E. Spyrou, D. Sgouropoulos, T. Giannakopoulos, P. Mylonas, ”Real-Time Arm Gesture Recognition Using 3D Skeleton Joint Data.”, 12(5):108, 2019, doi: https://doi.org/10.3390/a12050108

[7] C. Lugaresi, J. Tang, H. Nash, C. McClanahan, E. Uboweja, M. Hays, F. Zhang, C. Chang, M. Yong, J. Lee, W. Chang, W. Hua, M. Georg, M. Grundmann, ”MediaPipe: A Framework for Building Perception Pipelines”, 2019

[8] V. Bazarevsky, I. Grishchenko, K. Raveendran, T. Zhu, F. Zhang, M. Grundmann. ”BlazePose: On-device Real-time Body Pose tracking”, 2020

[9] S. Gasparrini, E. Cippitelli, S. Spinsante, E. Gambi. ”A Depth-Based Fall Detection System Using a Kinect® Sensor”, 2014. 14(2):2756-2775, doi: https://doi.org/10.3390/s140202756

[10] F. Destelle et al., ”Low-cost accurate skeleton tracking based on fusion of kinect and wearable inertial sensors,” 2014 22nd European Signal Processing Conference (EUSIPCO), 2014. pp. 371-375

[11] T. Q. Vinh, N. T. Tri, ”Hand gesture recognition based on depth image using kinect sensor”, 2015, 2nd National Foundation for Science and Technology Development Conference on Information and Computer Science (NICS), 2015. pp. 34-39, doi: 10.1109/NICS.2015.7302218

[12] OpenNI / NiTE2 official website. Available online on archive.org: https://web.archive.org/web/20140305034143/http://www.openni.org/files/nite/ (accessed on 4 July 2021)

[13] S. Wang, J. Song, J. Lien, I. Poupyrev, O. Hilliges, ”Interacting with Soli: Exploring Fine-Grained Dynamic Gesture Recognition in the Radio-Frequency Spectrum.”, In Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST ’16), 851–860, doi: https://doi.org/10.1145/2984511.2984565

[14] J. Lien, N. Gillian, M. Emre Karagozler, P. Amihood, C. Schwesig, E. Olson, H. Raja, I. Poupyrev, ”Soli: ubiquitous gesture sensing with millimeter wave radar”, July 2016, ACM Trans. Graph. 35, 4, Article 142, doi: https://doi.org/10.1145/2897824.2925953

[15] Specifications of the Google Next Hub device, a smart assistant with a display using the Soli sensor (https://store.google.com/us/product/nest_hub_2nd_gen_specs?hl=en-US)

[16] T. Yu, T. Kim, R. Cipolla, ”Unconstrained Monocular 3D Human Pose Estimation by Action Detection and Cross-Modality Regression Forest,” 2013 IEEE Conference on Computer Vision and Pattern Recognition, 2013, pp. 3642-3649, doi: 10.1109/CVPR.2013.467

[17] I. Culjak, D. Abram, T. Pribanic, H. Dzapo and M. Cifrek, ”A brief introduction to OpenCV”, 2012 Proceedings of the 35th International Convention MIPRO, 2012, pp. 1725-1730

[18] C. Shorten, T.M. Khoshgoftaar, ”A survey on Image Data Augmentation for Deep Learning”, 2019, J Big Data 6, 60, doi: https://doi.org/10.1186/s40537-019-0197-0

[19] V. Bazarevsky, Y. Kartynnik, A. Vakunov, K. Raveendran, M. Grundmann, “BlazeFace: Sub-millisecond Neural Face Detection on Mobile GPUs.”, 2019, ArXiv abs/1907.05047

[20] F. Zhang, V. Bazarevsky, A. Vakunov, A. Tkachenka, G. Sung, C. Chang, M. Grundmann, ”MediaPipe Hands: On-device Real-time Hand Tracking.”, 2020

[21] Accuracy score formula definition, SciKit-Learn official online documentation, https://scikit-learn.org/stable/modules/model_evaluation.html#accuracy-score